Efficiently Indexing High-Dimensional Data Spaces

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

von

Christian Böhm

Tag der Einreichung: 06.10.1998
Tag der mündlichen Prüfung: 17.12.1998

Berichterstatter:
Prof. Dr. Hans-Peter Kriegel, Ludwig-Maximilians-Universität München
Prof. Dr. Bernhard Seeger, Philipps-Universität Marburg

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. I extend my warmest thanks to my supervisor, 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. Bernhard Seeger who was readily willing to act as the second referee to this work. I would also like to thank Professor R. Bayer, Ph.D. for supporting me during my employment at the FORWISS institute, Technische Universität München. This work could not have grown and matured without the discussions with my colleagues. In particular I would like to thank Dr. Stefan Berchtold and Professor Dr. Daniel Keim from whom I learned important things about scientific work. During a research visit to AT&T, Florham Park, USA, I cooperated with Dr. H. V. Jagadish who inspired me to a great extent. Other fruitful discussions which brought this work forward took place with (in alphabetical order): Mihael Ankerst, Bernhard Braunmüller, Markus Breunig, Dr. Martin Ester, Andreas Miethsam, Jörg Sander, Thomas Schmidt, Dr. Thomas Seidl, and Dr. Xiaowei Xu. I thank them all. I would like to thank all my students who supported my work: Gerald Klump, Sven Messfeld, Urs Michel, Stefan Schönauer and Gert Unterhofer. Particular thanks go to Franz Krojer who took special care of our technical environment. This work could not have been completed without the background help of Susanne Grienberger. Besides shouldering much of the administrative burden, she carefully read this thesis and helped to polish the English. Last, but not least, I want to thank my parents, my friends and my girlfriend Bianca.

Christian Böhm
Munich, September 1998.

Abstract

Indexing high-dimensional data spaces is an emerging research domain. It gains increasing importance by the need to support modern applications by powerful search tools. In the so-called non-standard applications of database systems such as multimedia, CAD, molecular biology, medical imaging, time series processing and many others, similarity search in large data sets is required as a basic functionality.

A technique widely applied for similarity search is the so-called feature transformation, where important properties of the database objects are transformed into points of a multidimensional vector space, the so-called feature vectors. Thus, similarity queries are naturally translated into neighborhood queries in the feature space. In order to achieve a high performance in query processing, multidimensional index structures are used to manage the feature vectors. Unfortunately, multidimensional index structures deteriorate in performance when the dimension of the data space increases, because they are primarily designed for low-dimensional data spaces and due to a bunch of effects usually called the '*curse of dimensionality*'.

The general goal of this thesis is therefore the improvement of the efficiency of index-based query processing in high-dimensional data spaces.

For this purpose, a cost model for index-based query processing in high-dimensional data spaces was developed. It is applicable to a variety of index structures and query processing techniques and can be used for the evaluation of techniques and for optimization.

Based on this cost model, a variety of improvement and optimization techniques for multidimensional index structures was developed. The first, called DABS-tree, involves a cost model based split algorithm supporting a dynamic and local adaptation of the block size of the index structure. Dynamic block size adaptation is especially useful as we can show that conventional index structures often access data in too small portions.

The optimal unit of processing is not only dependent on the dimension but also on the number of objects currently stored in the database and on the distribution from which data and query points are taken.

A second technique for query processing based on our cost model is called tree striping. Here, the vectors are vertically decomposed into sub-vectors from data spaces of lower dimensionality. These subspaces are indexed and queried independently. The partial results of the single indexes must be merged to achieve a total result. Here, it is an optimization task to choose appropriate dimensionalities of the subspaces.

The next technique optimizes directly the shape of bounding boxes by exploiting *a priori* knowledge of the data set when bulk-load operations are applied. In this context, a new method for bottom-up construction of indexes is developed.

A further technique is intended for high-dimensional query processing in parallel environments. An optimal strategy for distributing points among servers in a network or among disks connected to a single computer is presented.

The last technique presented in this thesis is called Pyramid tree. It is based on a transformation of feature vectors and range queries into a one-dimensional space. For range queries using maximum metric, it turned out that this technique is not affected by the '*curse of dimensionality*'. Therefore, it can be efficiently used for indexing data spaces of very high dimensions.

All indexing and optimization techniques in this thesis were carefully analyzed. The practical impact was shown by exhaustive experimental evaluations yielding substantial performance improvements over state-of-the-art indexing techniques. The material presented in this thesis has matured the new research domain of indexing high-dimensional data spaces both theoretically as well as practically by a new cost model and various new index structures and optimization techniques for index-based query processing.

Abstract (In German)

Die Indexierung hochdimensionaler Datenräume ist eine neue Forschungsrichtung, die durch die Notwendigkeit, neue Anwendungen mit mächtigen Suchwerkzeugen auszustatten, zunehmend an Bedeutung gewinnt. In den sogenannten Nicht-Standard-Anwendungen von Datenbanksystemen wie z.B. Multimedia, CAD, Molekularbiologie, medizinische Bildverarbeitung, Analyse von Zeitreihen usw. ist die Ähnlichkeitssuche in großen Datenmengen eine wichtige Basisfunktion.

Eine weit verbreitete Technik zur Ähnlichkeitssuche ist die sogenannte Feature-Transformation, bei der wichtige Eigenschaften der Datenobjekte in Punkte eines mehrdimensionalen Vektorraums, die sogenannten Feature-Vektoren überführt werden. So werden Ähnlichkeitsanfragen auf natürliche Weise in Nachbarschaftsanfragen im Feature-Raum übersetzt. Um eine hohe Effizienz bei der Anfragebearbeitung zu erreichen, werden häufig multidimensionale Indexstrukturen zur Verwaltung der Feature-Vektoren eingesetzt. Leider versagen herkömmliche multidimensionale Indexstrukturen oft bei einer hohen Dimension des Datenraums, da sie in erster Linie für niedrigdimensionale Räume konzipiert wurden. Verantwortlich für das Versagen der Indexstrukturen sind eine Reihe von Effekten, die üblicherweise mit dem Begriff '*Fluch der hohen Dimension*' ('*Curse of Dimensionality*') belegt werden.

Das Hauptziel der vorliegenden Arbeit ist deshalb die Verbesserung der Performanz bei der indexbasierten Anfragebearbeitung in hochdimensionalen Datenräumen. Hierzu wurde ein Kostenmodell für die indexbasierte Anfragebearbeitung in hochdimensionalen Datenräumen entwickelt, das auf eine Reihe von Indexstrukturen und Techniken zur Anfragebearbeitung anwendbar ist und sowohl zur Evaluation dieser Techniken als auch zu deren Optimierung genutzt werden kann.

Basierend auf dem Kostenmodell wurden eine Menge von Optimierungstechniken entwickelt. Die erste, genannt *DABS-Tree* beinhaltet einen kostenmodellbasierten Split-

Algorithmus, der eine dynamische und lokale Adaptierung der logischen Blockgröße der Indexstruktur gestattet. Dies ist insbesondere nützlich, da in der Arbeit gezeigt wird, daß konventionelle Indexstrukturen häufig die Daten in zu kleinen Portionen einlesen. Die optimale Verarbeitungseinheit ist dabei nicht nur von Schemainformationen wie z.B. der Dimension des Datenraums abhängig sondern auch von Instanzinformationen wie z.B. der Anzahl von Objekten, die in der Datenbank gespeichert sind.

Eine weitere Optimierungstechnik, die auf dem Kostenmodell basiert, ist das sogenannte Tree-Striping. Die Vektoren werden hierbei in Teilvektoren mit niedrigerer Dimension zerlegt. Die Teilräume werden unabhängig voneinander indexiert und bearbeitet. Die Zwischenergebnisse der einzelnen Indexe werden am Schluß zu einem Endergebnis zusammengefaßt. Die Optimierungsaufgabe besteht bei dieser Technik in einer geeigneten Auswahl der Teilräume.

Die nächste Technik optimiert direkt die Form der Seitenregionen der Indexstruktur bei Vorliegen von *a-priori*-Wissen über die Datenobjekte. In diesem Zusammenhang wurde auch eine effiziente *Bottom-Up-Konstruktion* für Indexe entwickelt.

Eine weitere Technik dient zur hochdimensionalen Anfragebearbeitung in einer Parallelrechnerumgebung. Hier wurde eine optimale Strategie zur Verteilung der Datenpunkte auf verschiedene Knoten eines Rechnernetzes (das sogenannte *Declustering*) entwickelt.

Die letzte Technik die im Rahmen dieser Arbeit präsentiert wird, ist der sogenannte *Pyramid-Tree*. Für diese neuartige Indexstruktur wird gezeigt, daß sie für einen bestimmten Anfragetypus (*Range-Queries* in Verbindung mit der Maximumsmetrik) nicht dem '*Fluch der hohen Dimension*' unterworfen ist.

Alle neuentwickelten Indexierungstechniken wurden einer sorgfältigen theoretischen Analyse unterzogen. Ihre Praktikabilität wurde anhand einer umfassenden experimentellen Evaluation gezeigt, bei der das enorme Verbesserungspotential gegenüber bisherigen Techniken nachgewiesen werden konnte. Durch unsere Beiträge zur Kostenmodellierung und zahlreiche neue Indexstrukturen und Optimierungstechniken wurde das neue Forschungsgebiet der hochdimensionalen Indexierung daher sowohl um theoretische als auch praktische Aspekte substantiell bereichert.

Table of Contents

List of Figures

1 Introduction

2 Query Processing in High-Dimensional Data Spaces

3 A Cost Model for Query Processing in High-Dimensional Data Spaces

4 Dynamic Optimization of the Logical Block Size

5 Optimizing the Dimension Assignment

6 Optimizing the Geometry of Regions Using Bulk-Load Operations

7 Optimized Declustering for Parallel Query Processing

8 Indexing Ultra-High-Dimensional Feature Spaces

9 Conclusions

Chapter 1
Introduction

Information is the master key to economic success and influence in the contemporary society. It is generally agreed upon this proposition: “Only who can apply the newest information for his product development, is able to survive in the global competition” [Sch 95]. Crucial for the applicability of information is its quality and its fast availability. What is lacking most, however, is not the access to information resources but rather the facility to effectively and efficiently search for the required information.

If the structure of the information to be searched is simple, such as in one-dimensional numerical attributes or character strings, the problem can be considered as solved. Database management systems (DBMS) provide index structures for the management of such data [BM 77, Com 79] which are well-understood and widely applied.

In recent years, an increasing number of applications has emerged processing large amounts of complex, application-specific data objects [Jag 91, AFS 93, GM 93, FBFH 94, FRM 94, ALSS 95, Kor+ 96, BK 97, Ber 97, Kei 97, Sei 97]. In application domains such as multimedia, medical imaging, molecular biology, computer aided design, marketing and purchasing assistance, etc., a high efficiency of query processing is crucial due to the immense and even increasing size of current databases. The search in such databases, called non-standard databases, is seldom based on an exact match of objects. Instead, the search is often based on some notion of similarity which is also specific to the application.

We will start with a brief description of some of these new application domains, showing how similarity search is applied to fulfill the user’s requirements. Then we will

show a common solution to these different application domains, the so-called feature transformation. We will motivate that specialized index structures for high-dimensional vector spaces are needed for efficient query processing when using the feature approach. Unfortunately, the state-of-the-art in index structures and query processing techniques does not yield satisfactory performance. Based on this fact, we will substantiate our general motivation for the current thesis. An outline of the techniques proposed in this thesis will round off our introduction.

1.1 Non-Standard Applications to Database Systems

In this section, we will briefly sketch five applications to similarity search in database systems including the search for similar geometric shapes, as required in CAD databases, the search for similar color images in a multimedia database, medical imaging, the search for similar proteins in molecular biology, and the analysis of time sequences such as stock exchange rates, etc. Here, we do not strive for completeness, since this introduction is not actually a *related work* for our thesis. It should rather serve for motivation and illustration.

1.1.1 Retrieval of Similar Geometric Shapes

An important application domain for non-standard database systems is the area of Computer Aided Design (CAD). Most current CAD systems are file based systems which do not take advantage from any database technology. Some modern CAD systems currently use object-relational or object-oriented database technology, but only simple operations are supported, such as object retrieval according to the key. Database systems are in this case merely used as storage managers supporting data independence, concurrency and recovery, but not to support the user with a powerful search tool.

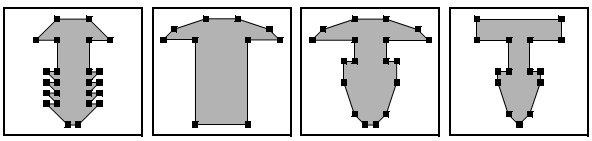


Figure 1: Similarity Search in CAD Databases

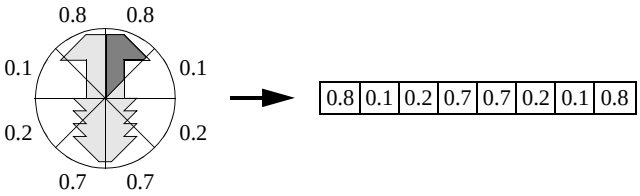


Figure 2: Section Coding

In a recent research project, the S3-System (*Similarity Search System*) was developed [Ber 97, BK 97, BKK 97]. The scope of this project was to reduce the diversity of parts in the car industry by providing the designers with a CAD database. The idea is to avoid the redesign of plastic clips when a similar design already exists. Cost can be saved by the reuse of mounting tools and injection moulds. The designs in the S3-project are two-dimensional. A further application to the search for similar geometric shapes is computer vision [Jag 91, GM 93, MG 93]. In different applications, the notion of similarity is defined differently. These definitions vary in their properties. Several similarity measures yield invariances which may be meaningful in some context, in another context not. The following invariances are considered as important: Translation invariance, rotation invariance, invariance with respect to uniform and non-uniform scaling, shearing invariance, invariance with respect to partial object occlusion. Moreover, we can distinguish between partial and total similarity. When speaking about total similarity, two objects have to be similar over all. In partial similarity, these objects have only to be similar in some detail.

Berchtold, Keim and Kriegel [Ber 97, BK 97, BKK 97] define similarity measures for two-dimensional polygons based on two different principles: The first, *section coding*, is based on volume coincidence. Starting from its center of gravity, the object is cut into slices in a pizza-like fashion (cf. figure 2). In each slice, the ratio of the volume intersection is determined independently. The vector of the ratios of all slices in the Euclidean space is used for the determination of the similarity. Seidl and Kriegel [Sei 97, KKS 98] extend the model by replacing the Euclidean metric by the more general quadratic form distance. By doing so, vicinity properties of two sectors can be taken into account which makes the model more realistic. Section coding is invariant with respect to translation, scaling and (to a limited degree) rotation.

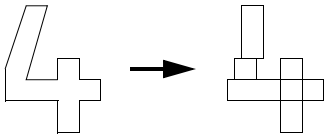


Figure 3: Object Transformation According to Jagadish [Jag 91]

The second similarity measure in the S3-project is based on the boundary of the polygon. The line segments of the polygon are transformed into a parametric form, the curvature. Then, an analytical Fourier transform is applied, and the coefficients are again interpreted as vectors in a Euclidean space. By applying this method to the complete object boundary, a measure for total similarity is defined. Partial similarity is defined by decomposing the object boundary into sequences of line segments with fixed length and applying the same technique to all sequences.

Jagadish proposes a technique for the retrieval of similar shapes in two dimensions [Jag 91]. He derives an appropriate object description from a rectilinear cover of an object, i.e. a cover consisting of axis-parallel rectangles (cf. figure 3). The rectangles belonging to a single object are sorted by size, and the largest ones serve as retrieval key for the shape of the object. Due to a normalization, invariance with respect to scaling and translation is achieved. The technique is not rotation-invariant.

Mehrotra and Gary suggest the use of boundary features for the retrieval of shapes [MG 93, MG 95, GM 93]. Here, a 2-D shape is represented by an ordered set of surface points, and fixed-sized subsets of this representation are extracted as shape features. All of these features are mapped to points in a multidimensional space. This method can handle translation, rotation and scaling invariance as well as partially occluded objects.

The QBIC (Query By Image Content) system [FBFH 94] contains a component for 2-D shape retrieval where shapes are given as sets of points. The method is based on algebraic moment invariants and is also applicable to 3-D objects [TC 91]. As an important advantage, the invariance of the feature vectors with respect to rigid transformations (translations and rotations) is inherently given. However, the adjustability of the method to specific application domains is restricted. From the available moment invariants appropriate ones have to be selected, and their weighting factors may be modified.

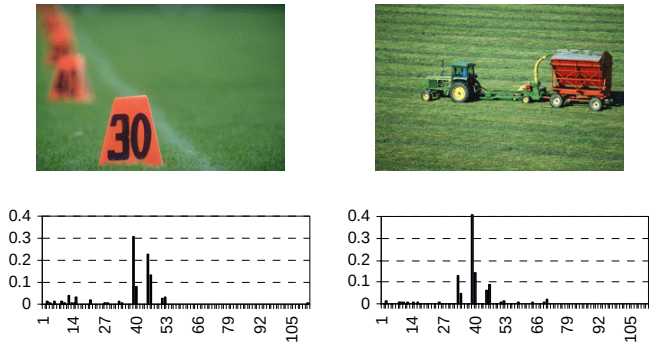


Figure 4: Two similar images and the corresponding 112-D color histograms [Sei 97].

1.1.2 Histogram-Based Similarity of Color Images

A natural way to search for color images in a multimedia database is based on color distributions [SH 94]. Two color images are defined to be similar if they contain approximately the same colors. This is formalized by the means of a color histogram. After accordingly reducing and normalizing the color spectrum of the images to a manageable number of different colors, the images are analyzed. For each color, the ratio of pixels is determined which are correspondingly colored (cf. figure 4).

An obvious way to compare color histograms is, again, to interpret them as vectors in Euclidean space. This approach leads to the difficulty that all pairs of different colors are interpreted as likewise dissimilar. In human perception, however, some colors are very similar to each other (e.g. red and orange) whereas others are very dissimilar (e.g. yellow and blue). The so-called cross-talk between similar colors can again be taken into account if not the Euclidean distance between the histogram vectors is determined, but the following quadratic form distance metric:

$$\delta_A^2(x, y) = (x - y) \cdot A \cdot (x - y)^T.$$

In this formula, the similarity matrix A contains the information which colors are similar to each other and to what degree. Both approaches, the QBIC system [FBFH 94] and the approach of Seidl and Kriegel [SK 97] use this definition of similarity in color images.

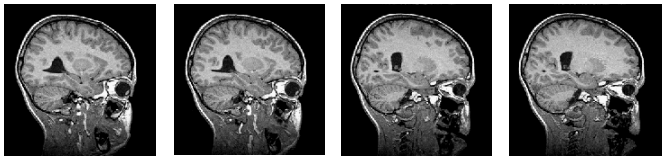


Figure 5: Medical Imaging (MRI) [Kei 97].

1.1.3 Medical Imaging

Korn et al. propose a method for searching similar tumor shapes in a medical image database [Kor+ 96]. For diagnostic purposes, especially the constitution of the surface of a tumor (parameters such as smoothness, raggedness, etc.) is important. Therefore, the similarity measure is in this method based on the theory of *Mathematical Morphology*, a quantitative theory of shape which incorporates a multi-scale component. In mathematical morphology, mappings are defined in terms of a structural element, a primitive shape such as a circle. It interacts with the input to transform it by two operations called opening and closing. Intuitively, opening is the set of points that a brush with the form of the structural element can reach when it is barely allowed to touch the boundary of the shape. In contrast, closing is equivalent to opening the complement of the object (cf. figure 6).

The similarity between two objects is defined in the following way: The objects are subject to a sequence of openings and closings with varying size of the structural element. For each opening (closing) in the sequence and for the original objects, the difference volume is determined. The largest observed difference volume determines the dissimilarity of the two objects.

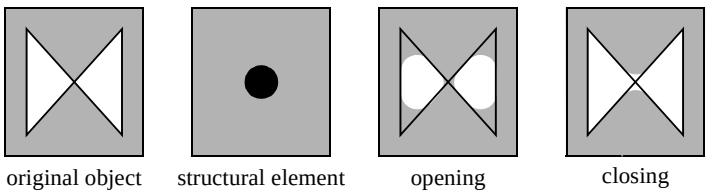


Figure 6: The Opening and Closing Operation of Mathematical Morphology.

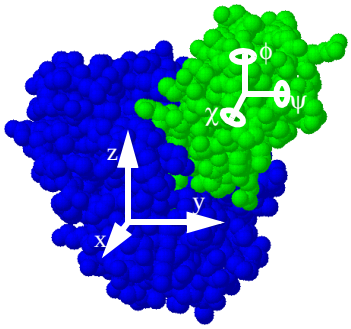


Figure 7: Example for Molecular Docking [Sei 97].

1.1.4 Molecular Biology

As in the area of CAD, multimedia and medical image processing, similarity queries are important in molecular biology [AGMM 90], too. Similarity queries are important since most of the biological functions in organisms are performed by the interaction of proteins. Similar functions are usually performed by molecules with a similar geometrical structure. There are various applications that require the three-dimensional structure of the molecular surface. The structure of molecules is provided by the Brookhaven Protein Data Bank which contains more than 3,000 molecules.

One of the most interesting tasks in molecular biology is the prediction of molecular interaction. Molecules interact if their surfaces have a complementary structure with respect to their 3-dimensional geometric shape and to electromagnetic and chemical properties. Finding molecules with a complementary structure, however, is a task closely related to the similarity search problem. The basic idea is to determine the complement of the query object and then to search for database objects which are similar to the complement of the query.

Kriegel, Schmidt and Seidl [KSS 97, KS 98] defined a similarity measure for segments of molecule surfaces which is based on fitting standard segments such as paraboloids to the molecular surface and determining the approximation error. The mutual approximation error is used as measure for the (dis-)similarity.

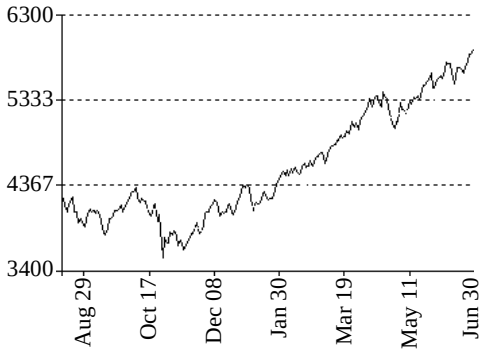


Figure 8: A Time Series: DAX Performance Index (Source: Frankfurt Stock Exchange).

1.1.5 Time Sequence Analysis

The analysis of time sequences has many applications in economic and other sciences. Questions of interest include, for example:

- Identify companies with similar growth patterns
- Determine products with similar selling patterns
- Discover stocks with similar movements in stock prices (cf. figure 8)
- Find if two musical scores are similar [AFS 93].

Agrawal et al. present a method for similarity search in a sequence database of one-dimensional data [AFS 93]. The authors define the square root of the sum of squared differences as the distance function between two sequences *x* and *y*:

$$\delta(x, y) = \sqrt{\sum_{0 \le t < n} (x_t - y_t)^2}$$

This definition coincides with the Euclidean distance of vectors and with the *energy* of the difference signal in a signal theoretic sense. The sequences are mapped onto points of a low-dimensional feature space by using a Discrete Fourier Transform.

The technique was later generalized for subsequence matching [FRM 94], and searching in the presence of noise, scaling, and translation [ALSS 95].

Further applications of similarity search include information retrieval [Kuk 92, Wel 71], vector quantization [RP 92] and data mining [AGGR 98, BJK 98, CD 97].

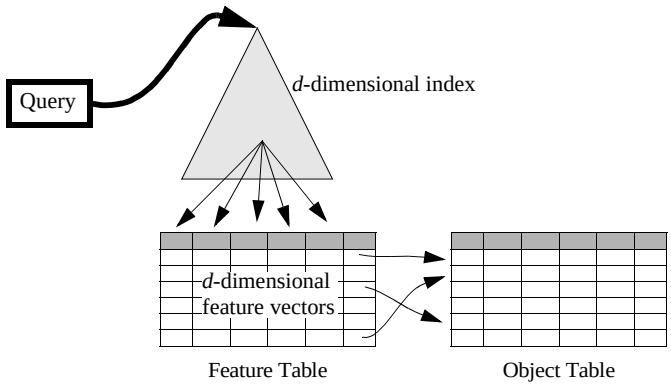


Figure 9: Multi-Step Processing of Similarity Queries.

1.2 Feature Transformation

At a first glance, the similarity notions of the five applications introduced above seem quite different from each other. Nevertheless, the similarity measures have some properties in common which facilitate query processing by the same paradigm in all these applications.

1.2.1 Object Distance

The first community of the similarity measures is that they all are defined in terms of a distance between two objects. That means, each similarity measure δ assigns a positive value to a pair of objects saying how dissimilar they are:

$$\delta : O \times O \rightarrow \Re_0^+.$$

Usually, the similarity measure δ is equal to 0 if and only if the two objects are identical. The higher δ is, the less similar are the two objects. Therefore, δ is also called the object distance. In all applications mentioned above, δ forms a metric, because it is positive, symmetric, and fulfills the triangle inequality. Recently, some query processing techniques have been proposed which can directly handle objects in a metric space [Yia 93, Chi 94, Uhl 91, Bri 95, BO 97, CPZ 97]. These structures, however, generally lack the required performance and were thus not applied in any of the sample applications.

1.2.2 Feature Distance

To handle similarity queries efficiently, usually a so-called feature transformation is applied. This approach extracts important properties from the objects in the database and transforms them into vectors of a *d*-dimensional vector space, the so-called feature vectors. Usually, the feature transformation is defined such that the distance between the feature vectors (the *feature distance*) either corresponds to the object distance or is, at least a lower bound thereof (“*lower bounding property*”). This way, the similarity search is naturally translated into a range query on the feature space.

The feature transformation is usually provided by an expert in the corresponding application domain, as it has to capture the most important and most distinguishing properties of the objects in order to achieve a good performance in query processing. In our example of time sequence databases, the discrete Fourier transform was used as feature transformation. In the example of medical image databases, the volumes of the object and its openings and closings were used as features. In molecular similarity, the features were based on the approximation by standard surfaces such as paraboloids. In all cases, the feature distance can be proven to be a lower bound of the object distance which is a necessary condition for the correctness of the method.

1.2.3 Multi-Step Query Processing

If the feature distance does not directly correspond to the object distance, but is only a lower bound, we talk about the paradigm of *multi-step query processing*. In a so-called *filter step*, a range query is processed on the feature space. As the feature distances are lower bounds of the actual object distances, the result of the range query is a set of candidates. It is guaranteed that each object satisfying the range query is contained in the candidate set (*no false dismissals*) but there may be candidates which are not actual answers to the similarity query. Therefore, the candidates have to be tested in the object space in a so-called *refinement step*. The paradigm yields advantages if only a few candidates have to be tested, i.e. if there is a good *filter selectivity.*

Figure 9 depicts the setting in multi-step query processing: The feature vectors are organized in an index. A query on this filter produces a set of candidates. The set is complete (no false dismissals), but may contain several objects which are not actual hits to the query. Therefore, the exact object representation must be loaded to the main memory. The final test whether an object is an actual answer to the query is called refinement step. From a database point of view, there are two main cost factors in this setting: The

cost for the filter step is mainly influenced by the quality of the index. The cost of refinement is mainly influenced by the filter selectivity, i.e. the size of the candidate set. As we assume the algorithm for the refinement step to be given by the application, we do not consider it as a parameter for optimization, although there may be potential for improvement, too. The filter step, however, is identical for any application. Hence, it is desired to support the filter step by the database management system.

This allows us to particularly focus on the following problem: Given a set N of d-dimensional points, how can we quickly search for points that fulfill a given query condition. The query condition could either be a multidimensional interval in which all points have to be located (*window query*) or it could be a point and we are looking for all points having a distance less than some value ε from this point (*range query*) or we are looking for the nearest neighbor of this point (*nearest neighbor query*). All these query types are useful in non-standard databases and it depends on the specific application which one will be used. In the following, we restrict our considerations on query processing in the feature space.

1.2.4 Index Structures

Various solutions to the problem of multidimensional search have been proposed. If the dimension d is sufficiently small, e.g. 3, we are able to use index structures such as the grid file [NHS 84], the hB-tree [LS 89, LS 90], the kd-tree [Ben 75, Ben 79] or the R^*-tree [BKSS 90]. However, if d is quite large, e.g. 16, these index structures do not provide an appropriate performance. The reasons for this degeneration of performance are subsumed by the term “curse of dimensionality”. The major problem in high-dimensional spaces is that most of the measures one could define in a d-dimensional vector space, such as volume, area, or perimeter are exponentially depending on the dimensionality of the space. Thus, many techniques work only in low-dimensional spaces where we still have an exponential dependency provided that the exponent is small enough.

To overcome these problems, a variety of specialized new index structures has been proposed in the past years dealing with the problem of high-dimensional indexing. Examples are the TV-tree [LJF 95], the SS-tree [WJ 96], or the X-tree [BKK 96]. For a complete overview cf. chapter 2. These structures, however, do not break the curse of dimensionality. They rather extend the area of dimensions where efficient indexing is possible, but still have their limitations when dimension increases to values above 20.

Unfortunately, the problems leading to the curse of dimensionality are complex. Therefore, no simple criterion exists do decide when to use which indexing method. To achieve good results in high-dimensional indexing, careful optimization must be undertaken. These optimizations are the most important motivation for the current thesis.

1.3 Outline of the Thesis

Chapter 2 is devoted to the related work. First, we introduce the common principles of the well-known index structures for high-dimensional data spaces and develop a framework to distinguish the previous approaches. We present the basic algorithms for query processing and index maintenance and describe then the state-of-the-art in high-dimensional indexing in a comprehensive way.

In chapter 3, we are going to introduce a cost model for query processing in high-dimensional data spaces. We start with a basic model for range queries and nearest neighbor queries which is applicable to query processing in low-dimensional data spaces under uniformity and independence assumption. We extend this model in two steps: First, we take the implications of high-dimensional query processing into account. This is done by a careful analysis of the effects and problems of high-dimensional data spaces. In a second step, the unrealistic assumption of a uniform and independent distribution of the data points is removed. For this purpose, we introduce the concept of the fractal dimension. We present all formulas of the cost model for the two most relevant vector metrics, the Euclidean metric (L_2) and the maximum metric (L_∞).

In chapter 4, we come to a first conclusion of the cost model presented in chapter 3. We use the cost model for the optimization of the logical block size of the index structure. As the optimum may dynamically change when new data objects are inserted in the database, and the optimum may also vary at different positions in the data space, the particularity of our approach is that the logical block size is adapted dynamically and independently in all data pages.

The next technique which is presented in chapter 5, is called tree striping. It is inspired from the so-called inverted list approach where not a single d-dimensional index is used for query processing, but a set of d one-dimensional indexes. Although the performance of inverted lists is very bad, it turns out that a mixture of inverted lists and multidimensional indexing outperforms both query processing techniques. In our approach, the vectors are decomposed into sub-vectors of a moderate dimensionality. The subspaces are

indexed and queried independently. The results of query processing have to be merged in a separate step. The decomposition decision is based on our cost model. Therefore the optimization task is in this chapter the right dimension assignment.

In the next chapter, we optimized the shape of the page regions under the assumption that the complete data set is previously known. In contrast to the classical approaches for low-dimensional indexing which tend to optimize for cube-like page regions, it can be derived from our cost model that cube-optimization is inappropriate when indexing high-dimensional data spaces. We can conclude that range search becomes more efficient when thin pages are cut from the borders of the data space. In the context of this chapter, a fast algorithm for the index construction from the scratch (bulk-load) was developed. The benefit is therefore two-fold: Additionally to the performance gain for the search operation, we present a sophisticated new algorithm for the fast index construction improving the efficiency of this operation by orders of magnitude.

Although these optimization techniques accelerate the range search and the nearest neighbor search in case of a moderate dimensionality by large factors, there still exists a dimension boundary where efficient index-based query processing is not possible. To overcome this problem, we propose in chapter 7 to exploit parallelism for high-dimensional query processing. We present an optimal declustering method. The general idea is to decompose the data space into quadrants and to assign the quadrants to servers such that neighboring quadrants are assigned to different servers. The quadrants can be represented as vertices in a graph, whereas the neighborhood relationships (we consider direct and indirect neighborhoods) are represented by the edges. Server assignment can be considered as graph coloring. An efficient solution, however, is possible, as not general graphs occur in our problem, but only a special type.

In chapter 8, we present an indexing technique for a special query type, range queries on maximum metric. It can be observed that it is for this special query type not subject to the curse of dimensionality. The ratio of page accesses is even decreasing with increasing dimension. The general idea of the technique is a decomposition of the data space in pyramid-like objects starting from the center of the data space. These pyramids are decomposed in a second step parallel to the base area. As every point can be represented by a pair containing the pyramid number and the height inside this pyramid, simple one-dimensional index structures can be applied for the management of the transformed points. Apart from the improved performance, a further advantage of the pyramid technique is that it is the easily to integrate in a relational database system.

Chapter 2
Query Processing in High-Dimensional Data Spaces

In this chapter, we will give an introduction about the basics of query processing in high-dimensional data spaces. We start with a few definitions which introduce important notions and formalize our problem description. Then, we will present the common principles of multidimensional index structures. There are two basic classes of multidimensional access methods: Hierarchical, data organizing structures such as R-trees [Gut 84, BKSS 90] and space organizing structures such as Multidimensional Hashing [HSW 88a, KS 86, KS 87, Oto 84] or grid-files [NHS 84, Fre 87, Hin 85, HSW 88b, KW 85, KS 88, Ouk 85]. For a comprehensive description of all multidimensional access methods, primarily concentrating on low-dimensional indexing problems, cf. to the survey of Gaede and Günther [GG 98]. We will concentrate here on the first class, the data organizing structures, since hashing-based methods do not play an important role in high-dimensional indexing. To our best knowledge, there exists no serious approach to solve the high-dimensional indexing problem with a space organizing structure. Also, we focus in this work on index structures primarily designed for secondary storage.

After introducing the common framework for multidimensional index structures, algorithms for query processing are presented according to all relevant query types. We will see that these algorithms can be expressed independently from the underlying multidimensional access method. In contrast, algorithms for the construction and maintenance of the index structures in a dynamic environment are specific to the corresponding

index structures and therefore presented later. Two algorithms for processing nearest neighbor queries are discussed in detail, because they are referenced later in this thesis.

In a related work section, we will give an overview over well-known index structures for high-dimensional query processing classifying the approaches by our common framework.

2.1 Basic Definitions

Before we are able to proceed, we need to introduce some notions and to formalize our problem description. In this section, we will define our notion of the database and we will develop a two-fold orthogonal classification for various neighborhood queries. Neighborhood queries can either be classified according to the metric which is applied to determine distances between points or according to the query type. Any combination between metrics and query types is possible.

2.1.1 Database

We assume that in our similarity search application, objects are feature-transformed into points of a vector space with a fixed, finite dimension *d*. Therefore, a database *DB* is a set of points in a *d*-dimensional data space *DS*. The data space *DS* is a subset of $\Re^d$. Usually, analytical considerations are simplified if the data space is restricted to the unit hypercube $DS = [0..1]^d$.

Our database is completely dynamic. That means, insertions of new points and deletions of points are possible and should be handled efficiently. The number of point objects currently stored in our database is abbreviated as *n*. We should note that the notion of a *point* is ambiguous. Sometimes, we mean a point object, i.e. a point stored in the database. In other cases, we mean a point in the data space, i.e. a position which is not necessarily stored in *DB*. The most common example for the second possibility is the query point. From the context, the intended meaning of the notion point will always be obvious.

Definition 1: Database

A database *DB* is a set of *n* points in a *d*-dimensional data space *DS*,

$$DB = \{P_0, ..., P_{n-1}\}$$

$$P_i \in DS, i = 0..n-1$$

$$DS \subseteq \mathbb{R}^d.$$

In some applications, objects cannot be mapped into feature vectors, however, there exists some notion of similarity between objects that can be expressed as a metric distance between objects. Thus, the objects are embedded in a metric space. These object distances can directly be used for query evaluation. Several index structures for pure metric spaces have been proposed [CPZ 97, Yia 93, Chi 94, Uhl 91, Bri 95, BO 97]. Our notion of a database, however, is restricted to vector spaces with finite dimension and therefore, we will not consider these approaches.

2.1.2 Vector Space Metrics

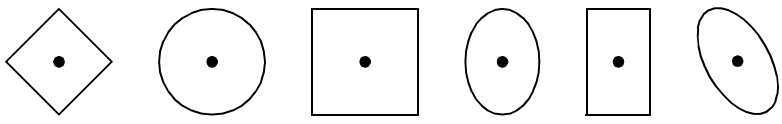
All neighborhood queries are based on the notion of the distance between two points P and Q in the data space. Depending on the application to be supported, several metrics to define the distances are applied. Most common is the Euclidean metric L_2 defining the usual Euclidean distance function δ_{em}:

$$\delta_{em}(P, Q) = \sqrt[2]{\sum_{i=0}^{d-1} (Q_i - P_i)^2}$$

But also other L_p metrics such as the Manhattan metric (L_1, also known as *city block metric*) or the maximum metric (L_∞) are widely applied:

$$\delta_{cm}(P, Q) = \sum_{i=0}^{d-1} |Q_i - P_i| \qquad \delta_{mm}(P, Q) = \max\{|Q_i - P_i|\}$$

Queries using the L_2 metric are (hyper-) sphere shaped. Queries using the maximum metric or the Manhattan metric are hypercubes and rhomboids, respectively (cf. figure



Manhattan (L_1) Euclidean (L_2) Maximum (L_∞) weighted Eucl. weighted Max. Ellipsoid

Figure 10: Metrics for Data Spaces.

10). If additional weights $w_0,..., w_{d-1}$ are assigned to the dimensions, then we define weighted Euclidean or weighted Maximum Metrics which correspond to axis-parallel ellipsoids and axis-parallel hyperrectangles:

$$\delta_{wem}(P, Q) = \sqrt[2]{\sum_{i=0}^{d-1} w_i \cdot (Q_i - P_i)^2} \qquad \delta_{wmm}(P, Q) = \max\{w_i \cdot |Q_i - P_i|\}$$

Arbitrarily rotated ellipsoids can be defined by using a positive definite similarity matrix *W*. This *quadratic form distance metric* is used for adaptable similarity search [Sei 97]:

$$\delta^2_{qfm}(P, Q) = (P - Q)^T \cdot W \cdot (P - Q)$$

2.1.3 Query Types

The first classification of queries is according to the vector space metric defined on the feature space. An orthogonal classification is based on the question whether the user defines a region of the data space or an intended size of the result set.

Point Query

The most simple query type is the point query. It specifies a point in the data space and retrieves all point objects in the database with identical coordinates:

$$\text{PointQuery}(\text{DB}, Q) = \{P \in \text{DB} \mid P = Q\}$$

A simplified version of the point query determines only the Boolean answer whether the database contains an identical point or not.

Range Query

In a range query, a query point *Q*, a distance *r*, and a metric *M* are specified. The result set comprises all points *P* from the database which have a distance smaller or equal to *r* from *Q* according to metric *M*:

Definition 2: Range Query

For a query object *Q*, a query range *r*, a metric *M* and a database DB, the range query retrieves the set

$$\text{RangeQuery}(\text{DB}, Q, r, M) = \{P \in \text{DB} \mid \delta_M(P, Q) \le r\}$$

Point queries can also be considered as range queries with a radius $r = 0$ and an arbitrary metric *M*. If *M* is the Euclidean metric, then the range query defines a hypersphere in the

data space from which all points in the database are retrieved. Analogously, the maximum metric defines a hypercube.

Window Query

A window query specifies a rectangular region in data space from which all points in the database are selected. The specified hyperrectangle is always parallel to the axis ("window"). We regard the window query as a region query around the center point of the window using a weighted maximum metric where the weights w_i represent the inverse of the side lengths of the window.

Nearest Neighbor Query

The range query and its special cases (point query and window query) have the disadvantage that the size of the result set is previously unknown. A user specifying the radius r may have no idea how many results his query may produce. Therefore, it is likely that he falls into one of two extremes: either he gets no answers at all or he gets almost all database objects as answers. To overcome this drawback, it is common to define similarity queries with a defined result set size, the nearest neighbor queries.

The classical nearest neighbor query returns exactly one point object as result which is the object with the lowest distance to the query point among all points stored in the database. The only exception from this one-answer rule is due to tie-effects. If several points in the database have the same (minimal) distance, then our first definition allows more than one answer:

Definition 3: Nearest Neighbor Query (Deterministic)

For a given query object Q and a given distance metric M, the deterministic nearest neighbor query retrieves the set:

$$\text{NNQueryDeterm}(\text{DB}, Q, M) = \{P \in \text{DB} \mid \forall P' \in \text{DB}: \delta_M(P, Q) \leq \delta_M(P', Q)\}$$

A common solution avoiding the exception to the one-answer rule uses non-determinism. If several points in the database have a minimal distance from the query point Q, an arbitrary point from the result set is chosen and reported as answer. We follow this approach:

Definition 4: Nearest Neighbor Query

For a given query object Q and a given distance metric M, a nearest neighbor query retrieves the set:

$$\text{NNQuery(DB, } Q, M) = \text{SOME}\{P \in \text{DB} \mid \forall P' \in \text{DB}: \delta_M(P, Q) \leq \delta_M(P', Q)\}$$

K-Nearest Neighbor Query

If a user does not only want one closest point as answer upon his query, but rather a natural number k of closest points, he will perform a *k-nearest neighbor query.* Analogously to the nearest neighbor query, the k-nearest neighbor query selects k points from the database such that no point among the remaining points in the database is closer to the query point than any of the selected points. Again, we have the problem of ties which can be solved either by non-determinism or by allowing more than k answers in this special case:

Definition 5: *k*-Nearest Neighbor Query

For a given query object Q and a given distance metric M, a k-nearest neighbor query retrieves the set:

$$\text{kNNQuery(DB, } Q, k, M) = \{P_0 \ldots P_{k-1} \in \text{DB} \mid \neg\exists P' \in \text{DB} \setminus \{P_0 \ldots P_{k-1}\} \wedge \neg\exists i, 0 \leq i < k: \delta_M(P_i, Q) > \delta_M(P', Q)\}$$

Approximate Nearest Neighbor Query

In *approximate nearest neighbor queries* and *approximate k-nearest neighbor queries*, the user also specifies a query point and a number k of answers to be reported. In contrast to *exact* nearest neighbor queries, the user is not interested exactly in the closest points, but wants only points which are not much farther away from the query point than the exact nearest neighbors. The degree of inexactness can be specified by an upper bound, how much farther away the reported answers may be compared to the exact nearest neighbors. The inexactness can be used for efficiency improvement of query processing.

Ranking Query

In a ranking query, the user specifies neither a range in the data space nor a result set size. Even though, the ranking query is more related to nearest neighbor queries than to range queries, because the first answer of a ranking query is always the nearest neighbor. The user has then the possibility to ask for further answers. Upon this request, the second nearest neighbor is reported, then the third and so on. The user decides after examining an answer if he needs further answers or not. Ranking queries can be especially useful in

the filter step of a multi-step query processing environment. Here, the refinement step usually takes the decision whether the filter step has to produce further answers or not.

2.1.4 Query Evaluation without Index

All query types introduced in the previous section can be evaluated by a single scan of the database. As we assume that our database is densely stored on a contiguous block on the secondary storage, all queries can be evaluated by a so-called *sequential scan* which is faster than the access of small blocks spread over wide parts of the secondary storage.

The sequential scan works as follows: The database is read in very large blocks determined by the amount of main memory available to query processing. After reading a block from disk, the CPU processes it and extracts the required information. After a block is processed, the next block is read in. We do not assume parallelism between CPU and disk I/O for any query processing technique presented in this thesis as our database server is single-threaded.

Further, we do not assume any additional information to be stored in the database. Therefore, the database has the following size in bytes:

$$\text{sizeof(DB)} = d \cdot n \cdot \text{sizeof(float)}$$

The cost of query processing based on the sequential scan is proportional to the size of the database in bytes.

2.2 Common Principles of High-Dimensional Indexing Methods

2.2.1 Structure

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

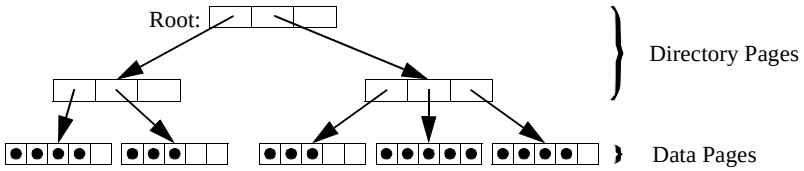


Figure 11: Hierarchical Index Structures.

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.

2.2.2 Management

The high-dimensional access methods are designed primarily for the secondary storage. Data pages have a data page capacity $C_{\max,data}$, defining how many data vectors can be stored in a data page at most. Analogously, the directory page capacity $C_{\max,dir}$ gives an upper limit to the number of subnodes in each directory node. The original idea was to choose $C_{\max,data}$ and $C_{\max,dir}$ 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. How to choose properly this logical page size will be investigated in chapter 3 and 4. The logical page sizes for data and directory nodes are constant for most of the index structures presented in this chapter. The only exception are the X-tree and the DABS-tree. The X-tree defines a basic page size and allows directory pages to extend over multiples of the basic page size. This concept is called supernode (cf. section 2.4.3). The DABS-tree is an indexing structure giving up the requirement of a constant blocksize. Instead, an optimal blocksize is determined individually for each page during the creation of the index. This **D**y-

namic **A**daptation of the **B**lock **S**ize gives the DABS-tree which is presented in chapter 4, its name.

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}. For obvious reasons, the root is generally allowed to hurt 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:

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

2.2.3 Regions

For efficient query processing it is important that the data are 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, solid and convex subset of the data space without holes. For most index structures, 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.

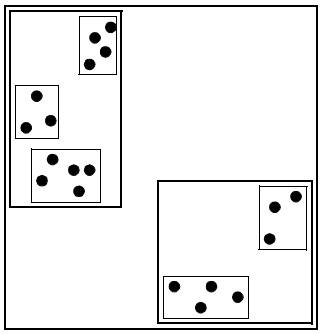


Figure 12: Corresponding Page Regions of an Indexing Structure.

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.

Both geometry and representation of the page regions must be optimized. If the geometry of the page region is suboptimal, the probability increases that the corresponding

page has to be accessed more frequently. If the representation of the region is unnecessarily large, the index itself gets larger yielding a worse efficiency in query processing as we will see later in this chapter.

2.3 Basic Algorithms

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.

2.3.1 Insert, Delete and Update

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 com-

pletely 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. The only exceptions are the LSD^h-tree [Hen 98] and the Space Filling Curves [Mor 66, FB 74, AS 83, OM 84, Fal 85, Fal 88, FR 89, Jag 90] (cf. section 2.4.5 and section 2.4.9). All other authors either suggest reinserting or do not provide a deletion algorithm at all. An alternative approach might be to permit underfilled pages and to maintain them until they are completely empty. The presence of delete operations and the choice of underflow treatment can affect $su_{\text{eff,data}}$ and $su_{\text{eff,dir}}$ positively as well as negatively.

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

2.3.2 Exact Match 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

```
bool ExactMatchQuery (Point q, PageAdr pa) {
        int i ;
        Page p = LoadPage (pa) ;
        if (IsDatapage (p) )
                for (i = 0 ; i < p.num_objects ; i ++)
                        if (q == p.object [i])
                                return true ;
        if (IsDirectoryPage (p) )
                for (i = 0 ; i < p.num_objects ; i ++)
                        if (IsPointInRegion (q, p.region[i]) )
                                if (ExactMatchQuery (q, p.sonpage[i]) )
                                        return true ;
        return false ;
}
```

Figure 13: Algorithm for Exact Match Queries.

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. Figure 13 contains the pseudocode for processing exact match queries.

2.3.3 Range Query

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 by using a weighted L_{max} metric.

```
PointSet RangeQuery (Point q, float r, Metric m, PageAdr pa) {
        int i ;
        PointSet result = EmptyPointSet ;
        Page p = LoadPage (pa) ;
        if (IsDatapage (p) )
                for (i = 0 ; i < p.num_objects ; i ++)
                        if (IsPointInRange (q, p.object [i], r, m)
                                AddToPointSet (result, p.object [i]) ;
        if (IsDirectoryPage (p) )
                for (i = 0 ; i < p.num_objects ; i ++)
                        if (RangeIntersectRegion (q, p.region[i]), r, m) )
                                PointSetUnion (result, RangeQuery(q, r, m, p.childpage[i])) ;
        return result ;
}
```

Figure 14: Algorithm for Range Queries.

The algorithm presented in figure 14 performs a recursive self-call for all child-pages whose corresponding page regions intersect with the query. The union of the results of all recursive calls is built and passed to the caller.

2.3.4 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 and their strengths and weaknesses are discussed.

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 algo-

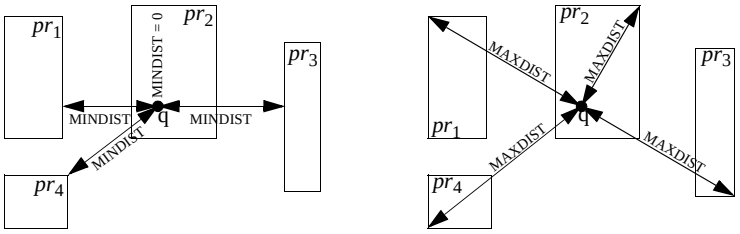


Figure 15: MINDIST and MAXDIST.

rithms 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.

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

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

Then, MINDIST is defined as follows:

Definition 6: MINDIST

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

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

An example of MINDIST is presented on the left side of figure 15. 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 indexing 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:

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

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 15. 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.

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:

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

where:

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

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 guaran-

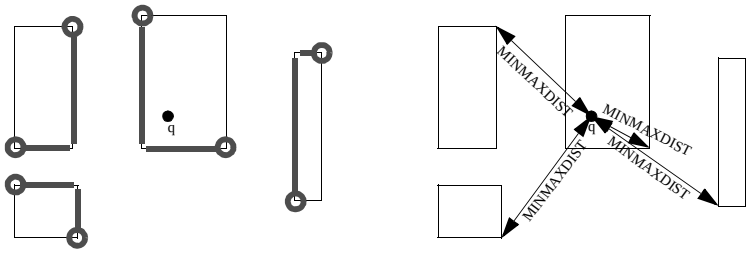


Figure 16: MINMAXDIST.

teed 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 16 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 16.

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 presented in figure 17 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

```
float pruning_dist/*      The current distance for pruning branches*/
      = INFINITE;/*       Initialization before the start of RKV_algorithm */
Point cpc ;         /*    The closest point candidate. This variable will contain
                          the nearest neighbor after RKV_algorithm has completed*/
void RKV_algorithm (Point q, Metric m, PageAdr pa) {
      int i ; float h ;
      Page p = LoadPage (pa) ;
      if (IsDatapage (p) )
            for (i = 0 ; i < p.num_objects ; i ++) {
                  h = PointToPointDist (q, p.object [i], m) ;
                  if (pruning_dist >= h) {
                        pruning_dist = h ;
                        cpc = p.object [i] ;
            }     }
      if (IsDirectoryPage (p) ) {
            sort (p, CRITERION) ; /* CRITERION is MINDIST or MINMAXDIST */
            for (i = 0 ; i < p.num_objects ; i ++) {
                  if (MINDIST (q, p.region[i]), m) <= pruning_dist)
                        RKV_algorithm (q, m, p.childpage[i] ) ;
                  h = MINMAXDIST (q, p.region[i]), m) ;
                  if (pruning_dist >= h)
                        pruning_dist = h ;
}     }     }
```

Figure 17: The RKV Algorithm for Finding the Nearest Neighbor.

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].

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.

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

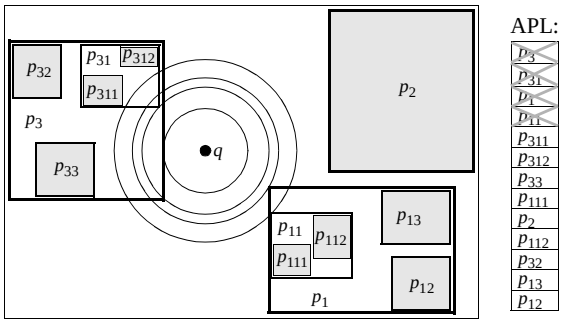


Figure 18: The HS Algorithm for Finding the Nearest Neighbor.

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.

Discussion

Now, we compare the two algorithms in terms of their space and time complexity. In the context of space complexity, we regard the available main memory as the most impor-

tant system limitation. We assume that the stack for recursion management and all priority queues are held in the main memory although one could also provide an implementation of the priority queue data structure suitable for secondary storage usage.

Lemma 1: Worst case space complexity of the RKV algorithm

The RKV algorithm has a worst case space complexity O (log *n*).

Proof (Lemma 1)

The only source of dynamic memory assignment in the RKV algorithm are the recursive calls of the function RKV_algorithm. The recursion depth is at most equal to the height of the indexing structure. The height of all high-dimensional index structures presented in this chapter is of the complexity O (log *n*). Since a constant amount of memory (one data or directory page) is allocated in each call, the claim of Lemma 1 follows.

❑

As the RKV algorithm performs a depth-first pass through the index structure, and no additional dynamic memory is required, the space complexity is O (log *n*). Lemma 1 is also valid for the *k*-nearest neighbor search if the additional space requirement for the closest point candidate list with a space complexity of O (*k*) is allowed for.

Lemma 2: Worst case space complexity of the HS algorithm

The HS algorithm has a space complexity of O (*n*) in the worst case.

Proof (Lemma 2)

The following scenario describes the worst case: Query processing starts with the root in APL. The root is replaced by its child nodes which are on the level *h* - 1 if *h* is the height of the index. All nodes on level *h* - 1 are replaced by their child-nodes, and so on, until all data nodes are in the APL. At this state, it is possible that no data page is excluded from the APL because no data point was encountered yet. The situation described above occurs, for example, if all data objects are located on a sphere around the query point. Thus, all data pages are in the APL and the APL is maximal because the APL grows only by replacing a page by its descendants. If all data pages are in the APL, it has a length of O (*n*).

❑

In spite of the order O (*n*), the size of the APL is only a very small fraction of the size of the data set because the APL contains only the page address and the distance between page region and query point *q*. If the size of the data set in bytes is *DSS*, then we have a number of *DP* data pages with

$$DP = \frac{DSS}{su_{\text{eff,data}} \cdot \text{sizeof(DataPage)}}.$$

Then, the size of the APL is *f* times the data set size:

$$\text{sizeof}(APL) = f \cdot DSS = \frac{\text{sizeof(float)} + \text{sizeof(address)}}{su_{\text{eff,data}} \cdot \text{sizeof(DataPage)}} \cdot DSS,$$

where a typical factor for a page size of 4 KBytes is $f = 0.3$ %, even shrinking with a growing data page size. Thus, it should be no practical problem to hold 0.3 % of a database in the main memory, although theoretically unattractive.

The complexity of the algorithm in terms of time is difficult to determine. We will develop the required methods in chapter 3. Comparing the two algorithms, we will prove optimality of the HS algorithm in the sense that it accesses as few pages as theoretically possible for a given index. We will further show that the RKV algorithm does not generally reach this optimum.

Lemma 3: Page regions intersecting the nearest neighbor sphere

Let *nndist* be the distance between the query point and its nearest neighbor. All pages that intersect a sphere around the query point having a radius equal to *nndist* (the so-called *nearest neighbor sphere*) must be accessed for query processing. This condition is necessary and sufficient.

Proof (Lemma 3)

(1) Sufficiency: If all data pages intersecting the nn-sphere are accessed, then all points in the database with a distance less than or equal to *nndist* are known to the query processor. No closer point than the nearest known point can exist in the database.

(2) Necessity: If a page region intersects with the nearest neighbor sphere but is not accessed during query processing, the corresponding subtree could include a point

that is closer to the query point than the nearest neighbor candidate. Therefore, accessing all intersecting pages is necessary.

❑

Lemma 4: Schedule of the HS algorithm.

The HS algorithm accesses pages in the order of increasing distance to the query point.

Proof (Lemma 4)

Due to the lower bounding property of page regions, the distance between the query point and a page region is always greater or equal to the distance of the query point and the region of the parent of the page. Therefore, the minimum distance between the query point and any page in the APL can only be increased or remain unchanged; never be decreased by the processing step of loading a page and replacing the corresponding APL entry. Since always the active page with minimum distance is accessed, the pages are accessed in the order of increasing distances to the query point.

❑

Lemma 5: Optimality of HS algorithm.

The HS algorithm is optimal in terms of the number of page accesses.

Proof (Lemma 5)

According to Lemma 4, the HS algorithm accesses pages in the order of increasing distance to the query point q. Let m be the lowest MINDIST in the APL. Processing stops if the distance of q to the *cpc* is less than m. Due to the lower bounding property, processing of any page in the APL cannot encounter any points with a distance to q less than m. The distance between the *cpc* and q cannot fall below m during processing. Therefore, exactly the pages with a MINDIST less or equal to the nearest neighbor distance are processed by the HS algorithm. According to Lemma 3, these pages must be loaded by any correct nearest neighbor algorithm. Thus, the HS algorithm yields an optimal number of page accesses.

❑

Now, we will demonstrate by an example that the RKV algorithm does not always yield an optimal number of page accesses. The main reason is that once a branch of the index has been selected, it has to be completely processed before a new branch can start. In the

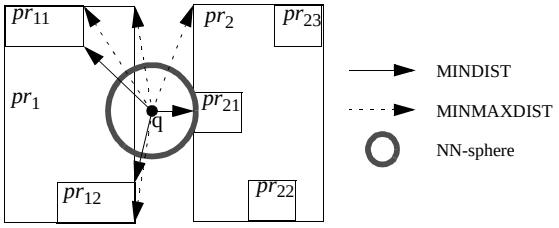


Figure 19: Schedules of RKV and HS Algorithm.

example of figure 19, both algorithms choose pr_1 to load first. Some important MINDISTs and MINMAXDISTs are marked in the figure with solid and dotted arrows, respectively. While the HS algorithm loads pr_2 and pr_{21}, the RKV algorithm has first to load pr_{11} and pr_{12}, because no MINMAXDIST estimate can prune the according branches. If pr_{11} and pr_{12} are not data pages, but represent further subtrees with larger heights, many of the pages in the subtrees will have to be accessed.

We have to summarize that the HS algorithm for nearest neighbor search is superior to the RKV algorithm when counting the page accesses. On the other side, it has the disadvantage of dynamically allocating main memory of the order O (n), although with a very small factor less than 1% of the database size. Additionally, the extension to the RKV algorithm for a k-nearest neighbor search is difficult to implement.

An open question is whether minimizing the number of page accesses will minimize the time needed for the page accesses, too. We will observe later that statically constructed indexes yield an inter-page clustering, meaning that all pages in a branch of the index are laid out contiguously on the background storage. Therefore, the depth-first search of the RKV algorithm could yield fewer disk-head movements than the distance-driven search of the HS algorithm. A new challenge could be to develop an algorithm for the nearest neighbor search directly optimizing the processing time rather than the number of page accesses.

2.3.5 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 ac-

cording 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, similarly as in Lemma 2, 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.

2.4 Previous Approaches to High-Dimensional Indexing

In this section, we will introduce and briefly discuss the most important index structures for high-dimensional data spaces. First, we will describe index structures using minimum bounding rectangles as page regions such as the R-tree, the R*-tree, and the X-tree.

We continue with the structures using bounding spheres such as the SS-tree and the TV-tree and conclude with two structures using combined regions. The SR-tree uses the intersection solid of MBR and bounding sphere as page region. The page region of a space filling curve is the union of not necessarily connected hypercubes.

Multidimensional access methods which have not been investigated for query processing in high-dimensional data spaces such as the R^+-tree [SSH 86, SRF 87], the hB-tree [LS 89, LS 90, Eva 94] or hashing-based methods [KS 86, KS 87, KS 88, Oto 84, NHS 84, Hin 85, HSW 88a, HSW 88b, KW 85, KS 88, Ouk 85, Fre 87] are excluded from our discussion. In the VAMSplit R-tree [JW 96] and in the Hilbert-R-tree [KF 94], methods for statically constructing R-trees are presented. Since these approaches are rather construction methods than indexing structures of its own, the presentation is delayed to chapter 6 where several construction methods are investigated.

2.4.1 R-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* comprises 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.

2.4.2 R*-tree

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, Beckmann, Kriegel, Schneider and Seeger 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].

2.4.3 X-tree

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 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 cases, a split should be avoided in order to avoid underfilled nodes.

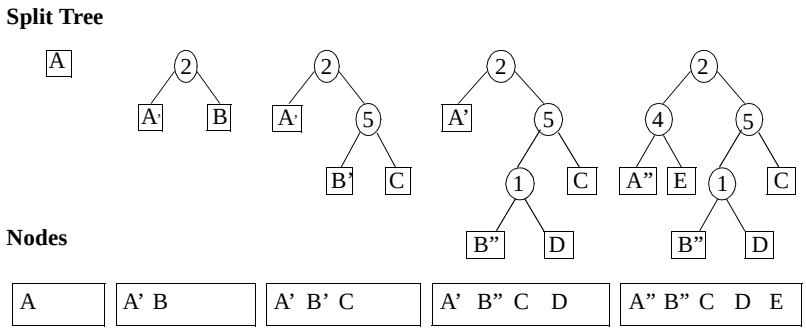


Figure 20: Example for the Split History.

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 20 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 20, it becomes clear that only

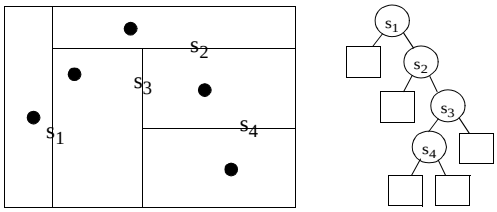


Figure 21: The kd-tree.

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 also 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.

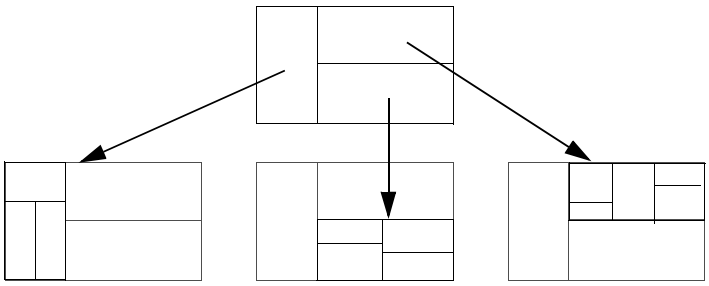


Figure 22: The k-d-B-tree.

2.4.4 k-d-B-tree

Like the R-tree and its variants, the k-d-B-tree [Rob 81] uses hyperrectangle shaped page regions. An adaptive *kd*-tree [Ben 75, Ben 79] is used for space partitioning (cf. figure 21). Therefore, complete and disjoint space partitioning is guaranteed. Obviously, the page regions are (hyper-) rectangles, but not minimum bounding rectangles. The general advantage of kd-tree based partitioning is that the decision which subtree is used is always unambiguous. The deletion operation is also supported in a better way than in R-tree variants because the leave nodes with a common parent exactly comprise a hyperrectangle of the data space. Thus, they can be merged without violating the conditions of complete and disjoint space partitioning.

Complete partitioning has the disadvantage that page regions are generally larger than necessary. Therefore, these pages are more often accessed during query processing than minimum bounding page regions. The second problem is that kd-trees usually are unbalanced. Therefore, it is not directly possible to pack contiguous subtrees into directory pages. The k-d-B-tree approaches this problem by a concept involving forced splits.

If some page has an overflow condition, it is split by an appropriately chosen hyperplane. The entries are distributed among the two pages and the split is propagated up the tree. Unfortunately, regions on lower levels of the tree may be also intersected by the split plane which must be split (*forced split*). As every region on the subtree can be affected, the time complexity of the insert operation is O (n) in the worst case. A minimum storage utilization guarantee cannot be provided. Therefore, theoretical considerations about the index size are difficult.

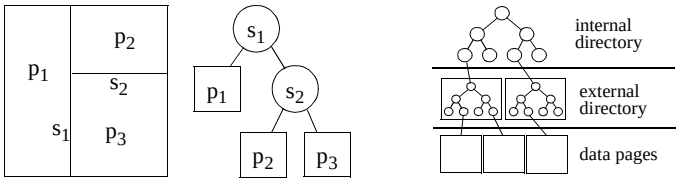


Figure 23: The LSD^h-tree.

2.4.5 LSD^h-tree

The directory of the LSD^h-tree [Hen 98, HSW 89] is also an adaptive *kd*-tree [Ben 75, Ben 79]. In contrast to R-tree variants and k-d-B-tree, the region description is coded in a sophisticated way leading to reduced space requirement for the region description. A specialized paging strategy collects parts of the *kd*-tree into directory pages. Some levels on the top of the *kd*-tree are assumed to be fixed in the main memory. They are called *internal directory* in contrast to the *external directory* which is subject to paging. In each node, only the split axis (e.g. 8 bit for up to 256-dimensional data spaces) and the position where the split-plane intersects the split axis (e.g. 32 bit for a float number) have to be stored. Two pointers to child nodes require 32 bit each. To describe k regions, $(k-1)$ nodes are required, leading to a total amount of $104 \cdot (k-1)$ bit for the complete directory. R-tree like structures require for each region description two float values for each dimension plus the child node pointer. Therefore, only the lowest level of the directory needs $(32 + 64 \cdot d) \cdot k$ bit for the region description. While the space requirement of R-tree directory grows linearly with increasing dimension, it is constant (theoretically logarithmic, for very large dimensionalities) for the LSD^h-tree. For 16-dimensional data spaces, R-tree directories are more than ten times larger than the corresponding LSD^h-tree directory.

The rectangle representing the region of a data page can be determined from the split planes in the directory. It is called the *potential data region* and not explicitly stored in the index.

One disadvantage of the *kd*-tree directory is that the data space is completely covered with potential data regions. In cases where major parts of the data space are empty, this results in performance degeneration. To overcome this drawback, a concept called *coded*

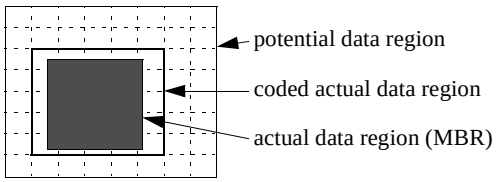


Figure 24: Region Approximation Using the LSD^h-tree.

actual data regions cadr is introduced. The *cadr* is a multidimensional interval conservatively approximating the *MBR* of the points stored in a data page. To save space in the description of the *cadr*, the potential data region is quantized into a grid of $2^{z \cdot d}$ cells. Therefore, only $2 \cdot z \cdot d$ bits are additionally required for each *cadr*. The parameter z can be chosen by the user. Good results are achieved by using a value of $z = 5$.

The most important advantage of the complete partitioning using potential data regions is that they allow a maintenance guaranteeing no overlap. The description page regions in terms of splitting planes forces the regions to be overlap-free, anyway. When a point has to be inserted into an LSD^h-tree, there exists always a unique *potential data region* in which the point has to be inserted. In contrast, the *MBR* of an R-tree may have to be enlarged for an insert operation which causes an overlap between data pages in some cases. A situation where no overlap-free enlargement is possible, is depicted in figure 25. The coded actual data regions may have to be enlarged during an insert operation. Since they are completely contained in a potential page regions an overlap cannot arise.

The split strategy for LSD^h-trees is rather simple. The split dimension is increased by one compared to the parent node in the *kd*-tree directory. The only exception from this

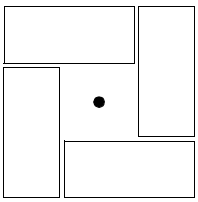


Figure 25: Situation in R-tree Variants where no Overlap-Free Insert is Possible.

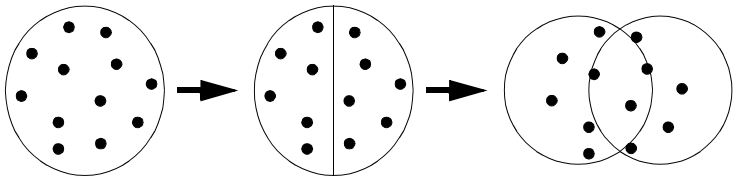


Figure 26: Situation in the SS-tree where no Overlap-Free Split is Possible.

rule is that a dimension having too few distinct values for splitting is left out of consideration.

As reported in [Hen 98], the LSD^h-tree shows a performance that is very similar to that of the X-tree, except the fact that inserts are done much faster in an LSD^h-tree because no complex computation takes place. Using a bulk-loading technique to construct the index, both index structures are equal in the performance. Also from an implementation point of view, both structures are of similar complexity: The LSD^h-tree has a rather complex directory structure and simple algorithms, whereas the X-tree has a rather straightforward directory and complex algorithms.

2.4.6 SS-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 26. 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.

2.4.7 TV-tree

The TV-tree [LJF 95] is designed especially for real data that are 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 27 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

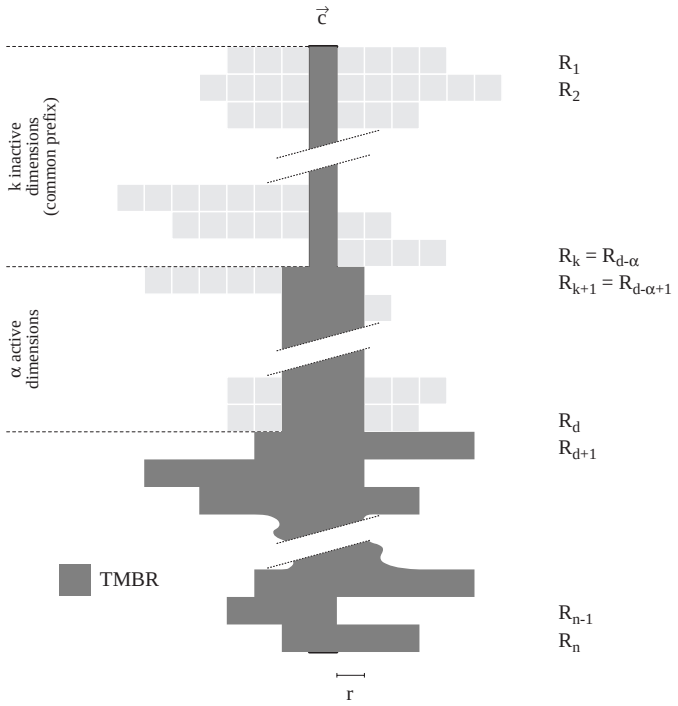


Figure 27: Telescope Vectors.

of telescope vectors increases the capacity of the directory pages. It was experimentally 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 maxi-

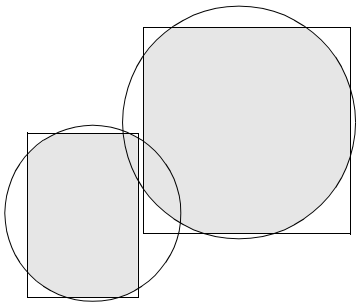


Figure 28: Page Regions of an SR-tree.

mum 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 for 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).

2.4.8 SR-tree

The SR-tree [KS 97] can be regarded as the combination of the R^*-tree and the SS-tree. It uses the intersection solid between a rectangle and a sphere as page region. The rectangular part is, like in R-tree variants, the minimum bounding rectangle of all points stored in the corresponding subtree. The spherical part is, like in the SS-tree, the minimum sphere around the centroid of the stored objects. Figure 28 depicts the resulting geometric object. Regions of SR-trees have the most complex description among all index structures presented in this chapter: It comprises $2d$ floating point values for the MBR and $d + 1$ floating point values for the sphere.

The motivation for using a combination of sphere and rectangle as presented by the authors is that according to the analysis presented in [WJ 96], spheres are basically better suited for processing nearest neighbor queries and range queries using the L_2-metric. We will present the theoretical framework for a more careful evaluation of this aspect in chapter 3. On the other hand, spheres are difficult to maintain and tend to produce much

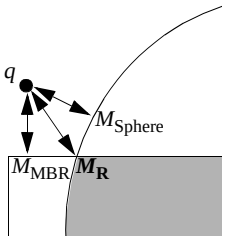


Figure 29: Incorrect MINDIST in the SR-tree.

overlap in splitting as depicted previously in figure 26. The authors believe therefore that a combination of R-tree and SS-tree will overcome both disadvantages.

The authors define the following function as the distance between a query point q and a region R:

$$\text{MINDIST}(q, R) = \max(\text{MINDIST}(q, R.\text{MBR}), \text{MINDIST}(q, R.\text{Sphere}))$$

This is not the correct minimum distance to the intersection solid, as depicted in figure 29 (which is drawn slightly too extreme, to make the problem visible): Both distances to the MBR and the sphere (meeting the corresponding solids at the points M_{MBR} and M_{Sphere}, respectively) are smaller than the distance to the intersection solid which is met in point M_R where the sphere intersects the rectangle. However, it can be shown that the above function MINDIST(q, R) is a lower bound of the correct distance function. Therefore, it is guaranteed that processing of range queries and nearest neighbor queries produces no false dismissals. But still, the efficiency can be worsened by the incorrect distance function. The MAXDIST function can be defined to be the minimum among the MAXDIST functions applied to MBR and sphere although a similar error is made as in the definition of MINDIST. Since no MINMAXDIST definition exists for spheres, the MINMAXDIST function for the MBR must be applied. This is also correct in the sense that no false dismissals are guaranteed but in this case no knowledge about the sphere is exploited at all. Some potential for performance increase is wasted.

Using the definitions above, range query processing and nearest neighbor query processing using both RKV algorithm and HS algorithm is possible.

Insert processing and split algorithm are taken from the SS-tree and only modified in a few details of minor importance. Additionally to the algorithms for the SS-tree, the MBRs have to be updated and determined after inserts and node splits. Information of

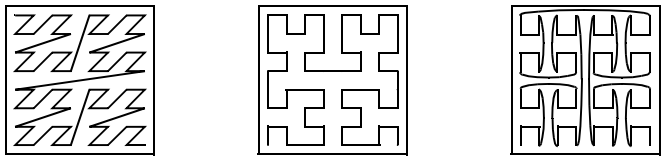


Figure 30: Examples of Space Filling Curves.

the MBRs is neither considered in the choice of branches nor in the determination of the split.

The reported performance results, compared to the SS-tree and the R^*-tree, suggest that the SS-tree outperforms both index structures. It is, however, open if the SR-tree outperforms the X-tree or the LSD^h-tree. No experimental comparison has been done yet to the best author's knowledge. Comparing the index structures indirectly by comparing both to the performance of the R^*-tree, we could draw the conclusion that the SR-tree does not reach the performance of the LSD^h-tree or the X-tree.

2.4.9 Space Filling Curves

Space filling curves [Sag 94] like Z-Ordering [Mor 66, FB 74, AS 83, OM 84], Gray Codes [Fal 85, Fal88] or the Hilbert Curve [FR 89, Jag 90] are mappings from a *d*-dimensional data space (original space) into a one-dimensional data space (embedded space). Using space filling curves, distances are not exactly preserved but points that are close to each other in the original space are likely to be close to each other in the embedded space. Therefore, these mappings are called distance-preserving mappings.

Z-Ordering is defined as follows: The data space is first partitioned into two halves of identical volume perpendicular to the d_0-axis.The volume on the side of lower d_0-values gets the name <0> (as a bit string), the other volume gets the name <1>. Then, each of the volumes is partitioned perpendicular to the d_1-axis, and the resulting sub-partitions of <0> get the names <00> and <01>, the sub-partitions of <1> get the names <10> and <11>, respectively. When all axis are used for splitting, d_0 is used for a second split, and so on. The process stops when a user-defined basic resolution *br* is reached. Then, we have a total number of 2^{br} grid cells, each with an individual bit string identified. If only grid cells with the basic resolution *br* are considered, all bit strings have the same

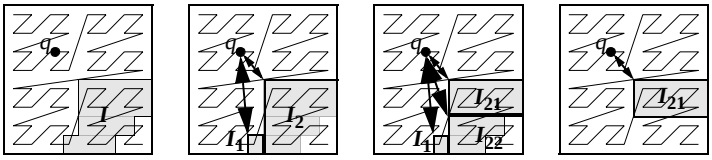


Figure 31: MINDIST Determination Using Space Filling Curves.

lengths, and can therefore be interpreted as binary representations of integer numbers. The other space-filling curves are defined similarly but the numbering scheme is slightly more sophisticated. This has been done in order to achieve that more neighboring cells get subsequent integer numbers. Some two-dimensional examples of space filling curves are depicted in figure 30.

Data points are transformed by assigning the number of the grid cell they are located in. Without presenting the details, we let SFC (p) be the function that assigns p to the corresponding grid cell number. Vice versa, $SFC^{-1}(c)$ returns the corresponding grid cell as a hyperrectangle. Then, any one-dimensional indexing structure capable of processing range queries can be applied for storing SFC(p) for every point p in the database. We assume in the following that a B^+-tree [Com 79] is used.

Processing of insert and delete operations and exact match queries is very simple because the points inserted or sought have merely to be transformed by the SFC function.

In contrast, range queries and nearest neighbor queries are based on distance calculations of page regions which have to be determined accordingly. In B-trees, before a page is accessed, only the interval $I = [lb\ ..\ ub]$ of values in this page is known. Therefore, the page region is the union of all grid cells having a cell number between lb and ub. The region of an index based on a space filling curve is a combination of rectangles. Based on this observation, we can define a corresponding MINDIST and analogously a MAXDIST function:

$$\text{MINDIST}(q, I) = \min_{lb \le c \le ub} \{\text{MINDIST}(q, \text{SFC}^{-1}(c))\}$$

$$\text{MAXDIST}(q, I) = \max_{lb \le c \le ub} \{\text{MAXDIST}(q, \text{SFC}^{-1}(c))\}$$

Again, no MINMAXDIST function can be provided because there is no minimum bounding property to exploit. The question is, how these functions can be evaluated efficiently without enumerating all grid cells in the interval [*lb* .. *ub*]. This is possible by splitting the interval recursively into two parts [*lb* .. *s*[and [*s* .. *ub*] where *s* has the form <*p*100...00>. Here, *p* stands for the longest common prefix of *lb* and *ub*. Then, we determine the MINDIST and the MAXDIST to the rectangular blocks numbered with the bit-strings <*p*0> and <*p*1>. Any interval having a MINDIST greater than the MAXDIST of any other interval or greater than the MINDIST of any terminating interval (see later) can be excluded from further consideration. The decomposition of an interval stops when the interval covers exactly one rectangle. Such an interval is called a terminal interval. MINDIST (*q*, *I*) is then the minimum among the MINDISTs of all terminal intervals. An example is presented in figure 31. The shaded area is the page region, a set of contiguous grid cell values *I*. In the first step, the interval is split into two parts I_1 and I_2, determining the MINDIST and MAXDIST (not depicted) of the surrounding rectangles. I_1 is terminal, because it comprises a rectangle. In the second step, I_2 is split into I_{21} and I_{22} where I_{21} is terminal. Since the MINDIST to I_{21} is smaller than the other two MINDIST values, I_1 and I_{22} are discarded. Therefore MINDIST (q, I_{21}) is equal to MINDIST (*q*, *I*).

A similar algorithm to determine MAXDIST (*q*, *I*) would exchange the roles of MINDIST and MAXDIST.

2.4.10 Summary

Table 1 shows the index structures described above in an overview. As the most important properties we identified the shape of the page regions, disjointedness and completeness and the most relevant decisions in the construction and maintenance of the index. As shape of the regions, we have rectangles, spheres and intersections and unions. If the rectangles are minimum bounding rectangles, this is marked in the table. Disjointedness means that the regions are not allowed to overlap. This is only guaranteed in the k-d-B-tree, the LSD^h-tree and in space-filling curves. Completeness means that the whole data space is covered with page regions. Also large empty parts of the data space are assigned to some data page. The LSD_h-tree has both, complete covering page regions and additionally page regions which are not extended over empty space. The criteria for choosing a subtree in performing an insert operation are based on proximity, volume, volume enlargement etc. The complete, and disjoint coverage of the data space with page regions

yields the advantage that the page where the point must be inserted, is always unique. Therefore, no heuristics must be applied. When more than one criterion is mentioned, the first has the highest weight, subsequent criteria are only applied if the first criterion yields a tie. The next row in the table summarizes the criteria for the choice of the split axis and the split plane. The last information in the table is if the index structure tries to perform a forced re-insert operation before splitting a page.

Name	Region	Disjoint	Complete	Criteria for Insert	Criteria for Split	Reinsert
R-tree	MBR	no	no	volume enlargement volume	(various algorithms)	no
R^*-tree	MBR	no	no	overlap enlargement volume enlargement volume	surface area overlap dead space coverage	yes
X-tree	MBR	no	no	overlap enlargement volume enlargement volume	split history surface/overlap dead space coverage	no
k-d-B-tree	rectangle	yes	yes	(unique due to complete, disjoint part.)	cyclic change of dim.	no
LSD^h-tree	kd-tree region	yes	no/yes	(unique due to complete, disjoint part.)	cyclic change of dim. # of distinct values	no
SS-tree	sphere	no	no	proximity to centroid	variance	yes
TV-tree	sphere with reduced dimension	no	no	# overlapping regions # inactive dimensions radius of region distance to center	seeds with least common prefix maximum distance	yes
SR-tree	intersect. sphere/ MBR	no	no	proximity to centroid	variance	yes
space filling curves	union of rectangles	yes	yes	(unique due to complete, disjoint part.)	according to space filling curve	no

Table 1: High-dimensional index structures and their properties

Chapter 3
A Cost Model for Query Processing in High-Dimensional Data Spaces

The aim of this chapter is to provide an introduction to the basic principles of cost modeling and to develop a cost model for high-dimensional query processing applicable to the R-tree like indexing structures. The basic principles presented in this chapter will be used later for modeling the performance of query processing techniques which are developed in this thesis.

There are various factors which have an influence on the performance of index based query processing. First of all the data set. The efficiency of index-based query processing depends on the dimension of the data space, the number of points in the database and on the data distribution from which the points are taken. Especially the correlation of dimensions is of high importance for the efficiency. Correlation means that the attributes of some dimensions are statistically not independent from each other. The value of one attribute is more or less determined by the values of one or more other attributes. In most cases, this dependency is not strict, but rather observable by the means of statistics. From a geometric point of view, correlation means that the data points are not spread over the complete data space. Instead, they are located on a lower-dimensional subset of the data space which is not necessarily a single linear subspace of the data space. There are indexing techniques which take profit from the fact that this actual dimension of the data

set is lower than the dimension of the data space. Other indexing techniques deteriorate in performance when a high correlation is inherent to the data set.

The metric for measuring the distance between two data points (Euclidean metric, Manhattan metric, maximum metric) has an important influence on the query performance, too.

A second set of influence factors is connected with the index structure. Most important is the shape of the page regions: It can be a rectangle, a sphere or a composed page region. If it is a rectangle, it can be a minimum bounding rectangle or is it part of a complete decomposition of the data space such as in the k-d-B-tree or in the LSD^h-tree. Most difficult to capture in a model are the various heuristics which are applied during insert processing and index construction. The impact of the heuristic on the volume and extension of page regions is very hard to quantify. A further influence factor is the layout of pages on the secondary storage. If the layout is clustered, i.e. pairs of adjacent pages are likely to be near by each other on disk, the performance can be improved if the query processing algorithm is conscious of this clustering effect.

A third set of influence factors is due to the choice of the query processing algorithm. As we pointed out in chapter 2, the HS algorithm a yields better performance in terms of page accesses than the RKV algorithm. Disk clustering effects can be exploited by algorithms considering the relative positions of pages on the background storage.

The outline of this chapter is as follows: After reviewing some related work on cost models, we start with the introduction of modeling range queries assuming an independent and uniform distribution of the data points. Moreover, we assume in the beginning that queries do not touch the boundary of the data space. Range queries are transformed into equivalent point queries by accordingly adapting the page regions. The central concept for this compensating adaptation is called *Minkowski sum* or *Minkowski enlargement*. We determine from the Minkowski sum the access probability of pages and use this access probability to develop an expectation for the number of page accesses. In the next section, nearest neighbor queries evaluated by the HS algorithm are modeled. This is conceptually done by a reduction step which transforms nearest neighbor queries into an equivalent range query. The corresponding range r can be estimated by a probability density function $p(r)$ using r as variable.

The simplifying assumptions of a uniform and independent data distribution and the ignorance of data space boundaries will be dropped step by step in the subsequent sec-

tions. First, the so-called boundary effects are introduced and shown to be important in high-dimensional data spaces. Our model is modified to take boundary effects into account. Then, we consider non-uniform data distributions which are independent in all dimensions. As the last step, we formalize correlations by the means of the *fractal dimension* and integrate this concept into our cost models for range queries and nearest neighbor queries.

In the last section, we will show, how the number of page accesses corresponds to the processing time of query processing. For this purpose, we introduce a model for data access in large, independent blocks from secondary storage (disk drive modeling).

Due to the high variety of cost estimates we will develop in this chapter the notions and the identifiers we have to use are complex. There are a few general identifiers for basic cost measures which will be used throughout this chapter:

V	for some volume
R	for the expected distance between a query point and its nearest neighbor
P	for some probability distribution function
X	for the access probability of an individual page
A	for the expectation of the number of page accesses.

The boundary conditions for the corresponding cost measure such as the distance metric and basic assumptions such as uniformity or independence in the distribution of data points are marked by subscripted indices of the general identifiers. For instance, $X_{\text{nn,em,ld,ui}}$ means the access probability for a nearest neighbor query (nn) using the Euclidean metric (em) on a low-dimensional data space (ld) under uniformity and independence assumption (ui). We distinguish:

- the query type: range query (r), nearest neighbor (nn) and *k*-nearest neighbor (knn)
- the applied metric: Euclidean metric (em) and maximum metric (mm)
- the assumed dimensionality: low-dimensional (ld) and high-dimensional (hd)
- the data distribution assumption: uniform/independent (ui), correlated (cor).

Especially the metric identifier (em or mm) is sometimes left out if an equation is valid for all applied metrics. In this case, all terms lacking the metric specification are understood to be substituted by the same metric, of course. The volume V is specified by a few indices indicating the geometric shape of the volume. Basic shapes are the hyper-sphere (s), the hyper-cube (c) and the hyper-rectangle (r). The Minkowski sum (cf. section 3.2) of two solids o_1 and o_2 is marked by a plus symbol ($o_1 \oplus o_2$). The clipping operation (cf. section 3.4) is marked by the intersection symbol ($o_1 \cap o_2$).

3.1 Review of Related Cost Models

Due to the high practical relevance of nearest neighbor queries, cost models for estimating the number of necessary page accesses have already been proposed several years ago. The first approach is the well-known cost model proposed by Friedman, Bentley and Finkel [FBF 77] for nearest neighbor query processing using maximum metric. The original model estimates leaf accesses in a kd-tree, but can be easily extended to estimate data page accesses of R-trees and related index structures. This extension was published in 1987 by Faloutsos, Sellis and Roussopoulos [FSR 87] and with slightly different aspects by Aref and Samet in 1991 [AS 91], by Pagel, Six, Toben and Widmayer in 1993 [PSTW 93] and by Theodoridis and Sellis in 1996 [TS 96]. The expected number of page accesses in an R-tree is

$$A_{\text{nn,mm,FBF}} = \left(\sqrt[d]{\frac{1}{C_{\text{eff}}}} + 1 \right)^d .$$

The assumptions of the model, however, are unrealistic for nearest neighbor queries on high-dimensional data for several reasons. First, the number N of objects in the database is assumed to converge to the infinity. Second, effects of high-dimensional data spaces and correlations are not considered by the model. Cleary [Cle 79] extends the model of Friedman, Bentley and Finkel [FBF 77] by allowing non-rectangular page regions, but still does not consider boundary effects and correlations. Eastman [Eas 81] uses the existing models for optimizing the bucket size of the kd-tree. Sproull [Spr 91] shows that the number of data points must be exponential in the number of dimensions for the models to provide accurate estimates. According to Sproull, boundary effects significantly contribute to the costs unless the following condition holds:

$$N >> C_{eff} \cdot \left(\sqrt[d]{\frac{1}{C_{eff} \cdot V_S\left(\frac{1}{2}\right)}} + 1 \right)^d$$

where $V_S(r)$ is the volume of a hypersphere with radius r which can be computed as

$$V_S(r) = \frac{\sqrt{\pi}^d}{\Gamma(d/2 + 1)} \cdot r^d$$

with the gamma-function $\Gamma(x)$ which is the extension of the faculty operation into the domain of real numbers: $\Gamma(x+1) = x \cdot \Gamma(x)$, $\Gamma(1) = 1$ and $\Gamma(\frac{1}{2}) = \sqrt{\pi}$.

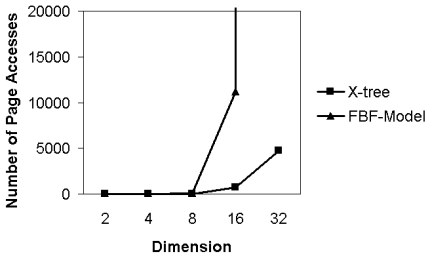


Figure 32: Evaluation of the model of Friedman, Bentley and Finkel.

Unfortunately, Sproull still assumes for his analysis uniformity and independence in the distribution of data points and queries.

The assumptions made in the existing models do not hold in the high-dimensional case. The main reason for the problems of the existing models is that they do not consider boundary effects. *"Boundary effects"* stands for an exceptional performance behavior, when the query reaches the boundary of the data space. As we show later, boundary effects occur frequently in high-dimensional data spaces and lead to pruning of major amounts of empty search space which is not considered by the existing models. To examine these effects, we performed experiments to compare the necessary page accesses with the model estimates. Figure 32 shows the page accesses versus the estimates of the model of Friedman, Bentley and Finkel. For high-dimensional data, the model completely fails to estimate the number of page accesses.

The basic model of Friedman, Bentley and Finkel has been extended in two different directions. The first is to take correlation effects into account by using the concept of the fractal dimension [Man 77, Sch 91]. There are various definitions of the fractal dimension which all capture the relevant aspect (the correlation), but are different in the details, how the correlation is measured. We will not distinguish between these approaches in our subsequent work.

Faloutsos and Kamel [FK 94] used the *box-counting fractal dimension* (also known as the *Hausdorff fractal dimension*) for modeling the performance of R-trees when processing range queries using maximum metric. In their model they assume to have a correlation in the points stored in the database. For the queries, they still assume a uniform and independent distribution. The analysis does not take into account effects of

high-dimensional spaces and the evaluation is limited to data spaces with dimensions less or equal to 3. Belussi and Faloutsos [BF 95] used in a subsequent paper the fractal dimension with a different definition (the *correlation fractal dimension*) for the selectivity estimation of spatial queries. In this paper, range queries in low-dimensional data spaces using Manhattan metric, Euclidean metric and maximum metric were modeled. Unfortunately, the model only allows the estimation of selectivities. It is not possible to extend the model in a straightforward way to determine expectations of page accesses.

Papadopoulos and Manolopoulos used the results of Faloutsos and Kamel and the results of Belussi and Faloutsos for a new model published in a recent paper [PM 97]. Their model is capable of estimating data page accesses of R-trees when processing nearest neighbor queries in a Euclidean space. They estimate the distance of the nearest neighbor by using the selectivity estimation of Belussi and Faloutsos [BF 95] in the reverse way. We will point out in section 3.3 that this approach is problematic from a statistical point of view. As it is difficult to determine accesses to pages with rectangular regions for spherical queries, they approximate query spheres by minimum bounding and maximum enclosed cubes and determine upper and lower bounds of the number of page accesses in this way. This approach makes the model inoperative for high-dimensional data spaces, because the approximation error grows exponentially with increasing dimension. A further asset of the model of Papadopoulos and Manolopoulos is that queries are no longer assumed to be taken from a uniform and independent distribution. Instead, the authors assume that the query distribution follows the data distribution.

The concept of fractal dimension is also widely used in the domain of spatial databases, where the complexity of stored polygons is modeled [Gae 95, FG 96]. These approaches are of minor importance for point databases.

The second direction, where the basic model of Friedman, Bentley and Finkel needs extension, are the boundary effects occurring when indexing data spaces of higher dimensionality.

Arya, Mount and Narayan [AMN 95, Ary 95] presented a new cost model for processing nearest neighbor queries in the context of the application domain of vector quantization. Arya, Mount and Narayan restricted their model to the maximum metric and neglected correlation effects. Unfortunately, they still assume that the number of points is exponential with the dimension of the data space. This assumption is justified in their application domain, but it is unrealistic for database applications.

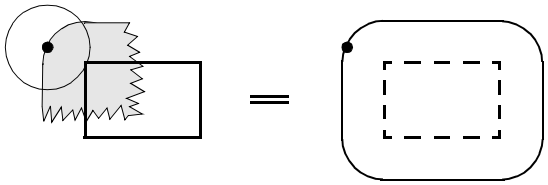


Figure 33: The Minkowski Sum.

Berchtold, Böhm, Keim and Kriegel [BBKK 97] presented in 1997 a cost model for query processing in high-dimensional data spaces. It provides accurate estimates for nearest neighbor queries and range queries using the Euclidean metric and considers boundary effects and avoids the disadvantages. To cope with correlation, the authors propose to use the fractal dimension without presenting the details. The main limitation of the model are (1) that no estimate for the maximum metric is presented, (2) that the number of data pages is assumed to be a power of two and (3) that a complete, overlap-free coverage of the data space with data pages is assumed. Weber, Schek and Blott [WSB 98] use the cost model by Berchtold et al. without the extension for correlated data to show the superiority of the sequential scan in sufficiently high dimensions. They present the VA-file, an improvement of the sequential scan.

In contrast to previous publications, the goal of this chapter is to present the basic principles of cost estimation and to derive cost models for various purposes. We derive models for both query types, range queries and nearest neighbor queries, and we present all formulas for maximum and Euclidean metric. We show how to cope with boundary effects, non-uniformity and correlation.

3.2 Range Query

In this section, we assume uniformity and independence in the distribution of both, data and query points. Moreover, we ignore the existence of a boundary of the data space or assume at least that page regions and queries are distant enough from the space boundary that the boundary is not touched. We start with a given page region and a given query range r and determine the probability with which the page is accessed, when the query point is assumed to be uniformly and independently chosen from the data space.

3.2.1 The Minkowski Sum

The corresponding page is accessed, whenever the query sphere intersects with the page region. To illustrate this, cf. figure 33. In all figures throughout this chapter, we will symbolize queries by spheres and page regions by rectangles. Also our notions (*"query sphere"*, for example) will often reflect this symbolization. We should note that queries using the maximum metric rather correspond to hypercubes than hyperspheres. We should further note that not all known index structures use hyperrectangles as page regions. Our concepts presented here are also applicable if the shape of the query is cubical or the shape of the page region is spherical.

We transform the range query into an equivalent point query by the following consideration: We call the center point of the range query the query anchor. Let us determine the set of all positions in the data space, from which the anchor must be taken such that the page is accessed. It is obvious from the diagram that the page region becomes enlarged by a sphere of the same radius r whose center point is drawn over the surface of the page region. As all marked positions are the positions of the query anchor, where the page is accessed, and as all unmarked positions are the positions of the query anchor, where the page is not accessed, the marked volume divided by the data space volume directly corresponds to the access probability of the corresponding page. As we assume for simplicity that the unit hypercube $[0..1]^d$ is the data space, the data space volume corresponds to 1.

For the determination of the volume, we have to distinguish various cases. The most simple case is that both volumes are hyperrectangles with side lengths a_i and b_i, for $0 \le i < d$, respectively. In this case, the volume of the Minkowski enlargement corresponds to the volume of the hyperrectangle with side lengths c_i, where each c_i corresponds to the sum of a_i and b_i:

$$V_{R \oplus R}((a_0, \ldots, a_{d-1}), (b_0, \ldots, b_{d-1})) = \prod_{0 \le i < d} (a_i + b_i).$$

If both volumes, query and region are hyperspheres with radius r_q and r_p, the Minkowski enlargement corresponds to a hypersphere with radius r_q+r_p. The corresponding volume of the hypersphere can be evaluated by the following formula:

$$V_{S \oplus S}(r_q, r_p) = \frac{\sqrt{\pi}^d}{\Gamma\left(\frac{d}{2}+1\right)} \cdot (r_q + r_p)^d.$$

The evaluation of the volume becomes complex if query and page region are differently shaped. In this case, every vertex of the hyperrectangle is enlarged by a part of a hypersphere. Every edge connecting two vertices of the hyperrectangle is enlarged by a certain part of a hypercylinder which is spherical in $d-1$ dimensions. A hyperrectangle has surfaces of various dimensionalities. Each surface with dimensionality k is enlarged by a part of a hypercylinder which is spherical in $d-k$ dimensions. In the remaining dimensions, the hypercylinder has the shape of the surface segment to which it is connected.

Before we determine the volume of the Minkowski enlargement in the most complex case of a hypersphere and a hyperrectangle, let us solve it for the simpler case that the rectangle is a hypercube with side length a. In this case, all k-dimensional surface segments have the volume a^k. Still open is the question, how many such surface segments exist and what the volume of the assigned part of the hypercylinder is. The number of surface segments can be determined by a combinatorial consideration. All surface segments (including the corners and the hyperrectangle itself) can be represented by a d-dimensional vector over the symbols 'L', 'U' and '*'. Here, the symbol 'L' stands for the *lower* boundary, 'U' for the *upper* boundary and the star stands for the complete edge connecting the lower and upper boundary. Using this notation, the vertices have no star, the edges have one star, the 2-dimensional surfaces have two stars in the vector, and so on. The hyperrectangle itself has d stars, no 'L' and no 'U' in its description vector. The number of k-dimensional surface segments corresponds to the number of different description vectors having k stars. The positions of the stars can be arbitrarily selected from d positions in the vector, yielding a binomial number of possibilities. The remaining d-k positions are filled with the symbols 'L' and 'U'. Therefore, the number of surface segments SSEGM(k) equals to:

$$\text{SSEGM}(k) = \binom{d}{k} \cdot 2^{d-k}.$$

The fraction of the hypercylinder at each surface segment is $1/2^{d-k}$. Therefore, we get the following formula for the Minkowski sum of a hypersphere with radius r and a hypercube with side length a:

$$V_{S \oplus C}(r, a) = a^d + \sum_{0 \le k < d} \text{SSEGM}(k) \cdot a^k \cdot \frac{1}{2^{d-k}} \cdot \frac{\sqrt{\pi^{d-k}}}{\Gamma\left(\frac{d-k}{2} + 1\right)} \cdot r^{d-k}$$

$$= \sum_{0 \le k \le d} \binom{d}{k} \cdot a^k \cdot \frac{\sqrt{\pi^{d-k}}}{\Gamma\left(\frac{d-k}{2} + 1\right)} \cdot r^{d-k}.$$

In the most complex case of non-cubical hyperrectangles, the *k*-dimensional surface segments must be summed up explicitly, which is a costly operation. Instead of the binomial multiplied with a^k, we have to summarize over all *k* combinations of the side lengths of the hyperrectangle:

$$V_{S \oplus R}(r, \vec{a}) = \sum_{0 \le k \le d} \left(\sum_{\{i_1 \ldots i_k\} \in 2^{\{0 \ldots d-1\}}} \left(\prod_{j=1}^{k} a_{i_j} \right) \right) \cdot \frac{\sqrt{\pi^{d-k}}}{\Gamma\left(\frac{d-k}{2} + 1\right)} \cdot (r)^{d-k}.$$

We should note that in most cases the determination of the Minkowski sum of a hypersphere and a hyperrectangle is an operation which is too costly, because it involves a number of basic operations (multiplications), which is exponential in the dimension. The explicit determination of the Minkowski sum of a real hyperrectangle and a hypersphere of high dimensionality must be avoided, even if exactness is sacrificed.

If the page region is a composition of rectangles such as in the approaches using space filling curves, it is also difficult to determine the volume of the Minkowski enlargement. It is possible to take the sum of the Minkowski enlargements of the elements of the composition and to use this sum as an upper bound. If the Minkowski enlargements of the elements overlap each other, this approach is not a good approximation. It is also very hard to provide a compensation for the approximation error, especially if the L_2 metric is applied, because the volume of the intersection among several spheres has to be estimated.

In both cases, where the determination of a Minkowski sum is difficult, a usual workaround is to transform the rectangle or the composition into a hypercube with equivalent volume. However, exactness is sacrificed in this approach, because the Minkowski sum of a non-cubical hyperrectangle is larger than the Minkowski sum of a volume-equivalent hypercube.

3.2.2 Estimating Rectangular Page Regions

The heuristics of R-tree variants such as the R*-tree and the X-tree strive to create page regions which yield low overlaps and are hypercube shaped. Therefore, our modeling approach assumes hypercubes as page regions. As we assume a uniform, independent data distribution, we can also assume that the volume of an arbitrarily selected region is directly proportional to the number of points enclosed in this region. We get the following proportionality law for hypercube page regions:

$$\frac{C_{\text{eff}}}{N} = \frac{V_{\text{R,nonbound}}}{V_{\text{DS}}} = \frac{a_{\text{nonbound}}^{d}}{1}.$$

It follows that

$$a_{\text{nonbound}} = \sqrt[d]{\frac{C_{\text{eff}}}{N}}.$$

In this formula for the side length of a typical page region, we assume a complete coverage of the data space with page regions. This assumption is not meaningful for minimum bounding rectangles. Usually, there is a small gap between two neighboring page regions. An expectation for the breadth of this gap under uniformity and independence assumption can be determined by projecting all points of a page onto the coordinate axis which is perpendicular to the gap. The average distance between two neighboring projections is obviously $1/C_{\text{eff}}$ times the side length of the region. This is also the expected value for the breadth of the gap by which the side length of the page region is decreased compared with a_{nobound}. Therefore, the side length of a minimum bounding rectangle can be estimated as:

$$a = \left(1 - \frac{1}{C_{\text{eff}}}\right) \cdot a_{\text{nobound}} = \left(1 - \frac{1}{C_{\text{eff}}}\right) \cdot \sqrt[d]{\frac{C_{\text{eff}}}{N}}.$$

The consideration of gaps between page regions is particularly important if the effective page capacity is low. Figure 34 shows the impact of the compensation factor on the volume of the page region. It shows the factor which decreases the volume of a page region when gaps are considered. The left diagram shows the compensation factor for a fixed dimension d=16 with varying C_{eff}. The strongest decrease occurs for low capacities. For a usual capacity between 20 and 40 points per data page, the compensation factor ranges between 40% and 70%. The right diagram shows the compensation factor

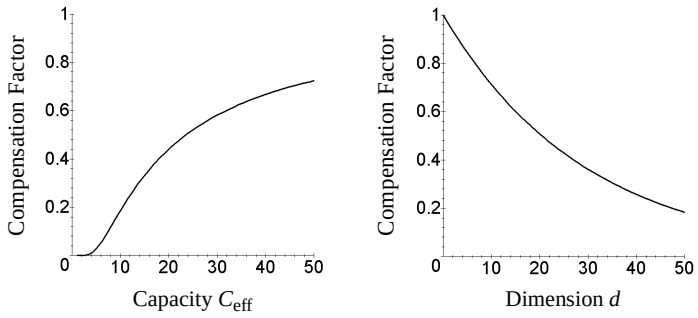


Figure 34: The Compensation Factor for Considering Gaps.

for a fixed effective page capacity (30 points) and varying dimension. Most compensation is necessary for large dimensions.

3.2.3 Expected Number of Page Accesses

By inserting the expected side length a into the formulas for the Minkowski enlargement, it is possible to determine the access probabilities of typical data pages under uniformity and independence assumption. This is for the maximum metric:

$$X_{\text{r,mm,ui}}(r) \;=\; (2r+a)^d \;=\; \left(2r + \left(1 - \frac{1}{C_{\text{eff}}}\right) \cdot \sqrt[d]{\frac{C_{\text{eff}}}{N}}\right)^d .$$

For Euclidean metric, the access probability for range queries with radius r evaluates to:

$$\begin{aligned} X_{\text{r,em,ui}}(r) \;&=\; \sum_{0 \le k \le d} \binom{d}{k} \cdot a^k \cdot \frac{\sqrt{\pi}^{d-k}}{\Gamma\left(\frac{d-k}{2}+1\right)} \cdot r^{d-k} \\ &=\; \sum_{0 \le k \le d} \binom{d}{k} \cdot \left(\left(1 - \frac{1}{C_{\text{eff}}}\right) \cdot \sqrt[d]{\frac{C_{\text{eff}}}{N}}\right)^k \cdot \frac{\sqrt{\pi}^{d-k}}{\Gamma\left(\frac{d-k}{2}+1\right)} \cdot r^{d-k}. \end{aligned}$$

From these access probabilities, the expected number of page accesses can be determined by multiplying the access probability with the number of data pages N / C_{eff}:

$$A_{\text{r,mm,ui}}(r) \;=\; \left(2r \cdot \sqrt[d]{\frac{N}{C_{\text{eff}}}} + 1 - \frac{1}{C_{\text{eff}}}\right)^d .$$

For Euclidean metric, the corresponding result is:

$$A_{\mathrm{r,em,ui}}(r) = \frac{N}{C_{\mathrm{eff}}} \cdot \sum_{0 \le k \le d} \binom{d}{k} \cdot \left(\left(1 - \frac{1}{C_{\mathrm{eff}}}\right) \cdot \sqrt[d]{\frac{C_{\mathrm{eff}}}{N}}\right)^k \cdot \frac{\sqrt{\pi^{d-k}}}{\Gamma\left(\frac{d-k}{2}+1\right)} \cdot r^{d-k}.$$

3.3 Nearest Neighbor Query

In chapter 2, the optimality of the HS algorithm for nearest neighbor search was proven. The HS algorithm yields exactly the same page accesses as an equivalent range query, i.e. a range query using the distance to the nearest neighbor as query range. This provides us with a concept to reduce the problem of modeling nearest neighbor queries to the problem of modeling range queries, which was solved in section 3.2. Therefore, we have to estimate the nearest neighbor distance.

Like in section 3.2 we start with the assumptions of an independent, uniform data distribution and we will ignore boundary effects. These effects will be investigated in depth in section 3.4 and section 3.5.

3.3.1 Coarse Estimation of the Nearest Neighbor Distance

A simple way to estimate the nearest neighbor distance is to choose a sphere in the data space such that an expected value of one data point is contained according to the current point density and to use the radius of this sphere as an approximation of the actual nearest neighbor distance. In the case of the maximum metric, we get the following formula:

$$\frac{1}{N} = V_q = (2r)^d \qquad r = \frac{1}{2\sqrt[d]{N}}.$$

For Euclidean metric, the corresponding formula is:

$$\frac{1}{N} = V_q = \frac{\sqrt{\pi^d}}{\Gamma(d/2+1)} \cdot r^d \qquad r = \sqrt[d]{\frac{\Gamma(d/2+1)}{N}} \cdot \frac{1}{\sqrt{\pi}}.$$

Unfortunately, this approach is not correct from the point of view of stochastics, because the operation of building an expectation is not invertible, i.e. the expectation of the radius cannot be determined from the expectation of the number of points in the corre-

sponding sphere. The approximation determined by this formula is rather coarse and can be used if a fast and simple evaluation is of higher importance than the accuracy of the model. The general problem is that even under uniformity and independence assumption the nearest neighbor distance yields a certain variance, when several range queries are executed.

3.3.2 Exact Estimation of the Nearest Neighbor Distance

A stochastically correct approach is to determine a distribution function for the nearest neighbor distance, and to derive an expectation of the nearest neighbor distance from the corresponding probability density function. From this probability density function, the expectation of the number of page accesses can also be derived.

The distribution function $P(r)$ determines the probability that the nearest neighbor distance is smaller than the variable r. The nearest neighbor distance is *larger than r* if and only if no data point is contained in the sphere with radius r. The event 'distance is larger than r' is the opposite of the event needed in our distribution function. Therefore, $P(r)$ is as follows:

$$P(r) = 1 - (1 - V(r))^N.$$

Due to the convergence of the limit

$$\lim_{\nu \to \infty} \left(1 + \frac{k}{\nu}\right)^{\nu} = e^k$$

the distribution function can be approximated for a large number of objects N by the following formula:

$$P(r) \approx 1 - e^{-N \cdot V(r)}.$$

This approximation yields negligible relative errors for a large N (starting from 100) and will facilitate the evaluation later.

For maximum metric and Euclidean metric, the probability distribution function $P(r)$ can be evaluated in the following way:

$$P_{mm}(r) = 1 - (1 - (2r)^d)^N,$$

$$P_{em}(r) = 1 - \left(1 - \frac{\sqrt{\pi^d}}{\Gamma(d/2 + 1)} \cdot r^d\right)^N.$$

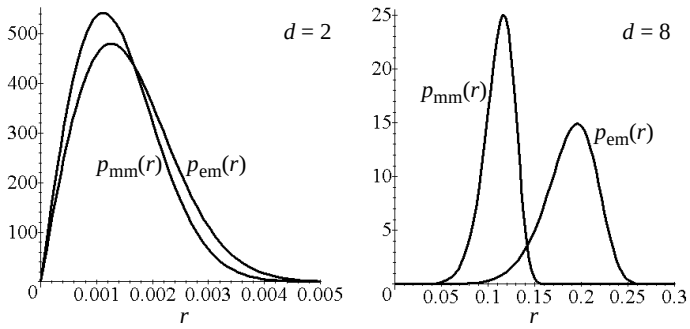


Figure 35: Probability Density Functions.

From the distribution function $P(r)$ a probability density function $p(r)$ can be derived by differentiation:

$$p(r) = \frac{\partial P(r)}{\partial r}.$$

For maximum and Euclidean metric, this evaluates to:

$$p_{\mathrm{mm}}(r) = \frac{\partial P_{\mathrm{mm}}(r)}{\partial r} = \frac{d \cdot N}{r} \cdot (1 - (2r)^d)^{N-1} \cdot (2r)^d,$$

$$p_{\mathrm{em}}(r) = \frac{\partial P_{\mathrm{em}}(r)}{\partial r} = \frac{d \cdot N}{r} \cdot \left(1 - \frac{\sqrt{\pi}^d}{\Gamma(d/2+1)} r^d\right)^N \cdot \frac{\sqrt{\pi}^d}{\Gamma(d/2+1)} r^d.$$

Figure 35 shows some probability density functions for 100,000 uniformly distributed data points in a two-dimensional and an 8-dimensional data space, respectively.

To determine the expected value of the nearest neighbor distance, the probability density function multiplied with r must be integrated from 0 to infinity:

$$R = \int_0^\infty r \cdot p(r) \partial r,$$

$$R_{\mathrm{mm}} = d \cdot N \cdot \int_0^\infty (1 - (2r)^d)^{N-1} \cdot (2r)^d \partial r,$$

$$R_{\text{em}} = d \cdot N \cdot \int_0^{\infty} \left(\left(1 - \frac{\sqrt{\pi}^d}{\Gamma(d/2+1)} r^d \right)^N \cdot \frac{\sqrt{\pi}^d}{\Gamma(d/2+1)} r^d \right) \partial r .$$

The integration variable is denoted by '∂' instead of the more usual 'd' to avoid confusion with the identifier 'd' standing for the dimension of the data space. The integral is not easy to evaluate analytically.

3.3.3 Numerical Evaluation

We present two numerical methods to evaluate the integral presented in this chapter numerically. The first is based on the binomial theorem. Basically, the probability density function $p(r)$ is a polynomial of the degree $d \cdot N$. It can be transformed into the coefficient form $p(r) = a_0 r^{d \cdot N} + a_1 r^{d \cdot N - 1} + \ldots$ using the binomial theorem. We demonstrate this for the maximum metric:

$$(1 - (2r)^d)^{N-1} = \sum_{0 \le i \le N-1} \binom{N-1}{i} \cdot (-1)^i \cdot (2r)^{d \cdot i};$$

$$p_{\text{mm}}(r) = \frac{d \cdot N}{r} \cdot (2r)^d \cdot \sum_{0 \le i \le N-1} \binom{N-1}{i} \cdot (-1)^i \cdot (2r)^{d \cdot i}.$$

This alternating series can be approximated with low error bounds by the first few summands ($0 \le i \le i_{\max}$) if the absolute value of the summands is monotonically decreasing. This is possible if the power of r decreases in a steeper way with increasing i than the binomial increases. This is guaranteed if the following condition holds:

$$\frac{N}{2} \le \frac{1}{(2r)^d}; \; r \le \frac{1}{2} \cdot \sqrt[d]{\frac{2}{N}}; \; p_{\text{mm}}(r) \approx \frac{d \cdot N}{r} \cdot (2r)^d \cdot \sum_{0 \le i \le i_{\max}} \binom{N-1}{i} \cdot (-1)^i \cdot (2r)^{d \cdot i}.$$

Therefore, we approximate our formula for the expected nearest neighbor distance in the following way:

$$R_{\text{mm}} \approx d \cdot N \cdot \int_0^{\left(\frac{1}{2} \cdot \sqrt[d]{\frac{2}{N}}\right)} \left((2r)^d \cdot \sum_{0 \le i \le i_{\max}} \binom{N-1}{i} \cdot (-1)^i \cdot (2r)^{d \cdot i} \right)$$

$$\approx \quad d \cdot N \cdot \sum_{0 \le i \le i_{\max}} \left(\begin{array}{c} N-1 \\ i \end{array} \right) \cdot (-1)^i \cdot \int_0^{\left(\frac{1}{2} \cdot \sqrt[d]{\frac{2}{N}}\right)} (2r)^{d \cdot (i+1)}$$

$$\approx \quad \sum_{0 \le i \le i_{\max}} \left(\begin{array}{c} N-1 \\ i \end{array} \right) \cdot \frac{(-1)^i}{i+1+\frac{1}{d}} \cdot \left(\frac{2}{N}\right)^{i+\frac{1}{d}}.$$

The same simplification can be applied for the Euclidean metric. However, an alternative way based on a histogram-approximation of the probability density function yields lower approximation errors and causes even a lower effort in the evaluation.

To facilitate numerical integration methods such as the middlebox approximation, the trapezoid approximation or the combined method according to Simpson's rule [PFTV 88], we must determine suitable boundaries, where the probability density function has values which are substantially greater than 0. If we consider for example figure 35, we observe that for d=8, $p_{\text{mm}}(r)$ is very close to 0 in the two ranges $0 \le r \le 0.05$ and $0.16 \le r \le \infty$. Only the range between the lower bound r_{lb}=0.05 and the upper bound r_{ub} contributes significantly. The criterion for a sensible choice of lower and upper bounds is based on the distribution function which corresponds to the area below the density function. We choose the lower bound r_{lb} such that the area in the ignored range $[0..r_{\text{lb}}]$ corresponds to a probability less than 0.1% and do the same for the upper bound r_{ub}. We get the following two conditions, resulting from the approximation of the distribution function:

$$P(r) \approx 1 - e^{-N \cdot V(r)} \ge 0.001 \qquad P(r) \approx 1 - e^{-N \cdot V(r)} \le 0.999$$

$$r \ge r_{\text{lb}} = V^{-1}\left(\frac{-\ln 0.999}{N}\right) \qquad r \le r_{\text{ub}} = V^{-1}\left(\frac{-\ln 0.001}{N}\right).$$

Integration can therefore be bounded to the interval from r_{lb} to r_{ub}. The integral can be approximated by a sum of trapezoids or by a sum of rectangles:

$$R = \int_0^{\infty} r \cdot p(r) \partial r$$

$$\approx \int_{r_{lb}}^{r_{ub}} r \cdot p(r) \partial r$$

$$\approx \frac{r_{ub} - r_{lb}}{i_{max}} \cdot \sum_{0 \le i < i_{max}} \left(\frac{r_{ub} - r_{lb}}{i_{max}} \cdot i + r_{lb} \right) \cdot p\left(\frac{r_{ub} - r_{lb}}{i_{max}} \cdot i + r_{lb} \right).$$

As we bound the integration to a small interval, a small number of rectangles or trapezoids is sufficient for a high accuracy. To achieve a relative error less than 1.0%, an approximation by $i_{max} = 5$ rectangles was required in our experiments.

3.3.4 K-Nearest Neighbor Query

The cost model developed in the previous sections can also be extended for estimating *k*-nearest neighbor queries. For the coarse model, this is straightforward since the volume is to choose such that *k* objects are contained rather than one. Therefore, the term $1/N$ must be replaced by k/N. For maximum metric, the estimated distance is:

$$R_{mm}(k) \approx \frac{1}{2} \cdot \sqrt[d]{\frac{k}{N}}.$$

For Euclidean metric, the result is analogous:

$$R_{em}(k) \approx \sqrt[d]{\frac{k \cdot \Gamma(d/2 + 1)}{N}} \cdot \frac{1}{\sqrt{\pi}}.$$

For the exact model, the probability distribution must be modeled as a summation of Bernoulli-chains with lengths ranging from 1 to *k*. The probability that at least *k* points are inside the volume $V(r)$ corresponds to the following formula:

$$P_k(r) = 1 - \sum_{0 \le i < k} \binom{n}{i} \cdot V(r)^i \cdot (1 - V(r))^{n-i}.$$

For $k = 1$, the formula corresponds to the distribution function $P(r)$ for nearest neighbor queries. The probability density function and the expectation of the nearest neighbor distance are determined analogously to section 3.3.2.

We should note that the peak in the probability density function of a *k*-nearest neighbor query becomes steeper with increasing *k* (decreasing variance). Therefore, the approximation by the coarse model which is bad for high, asymmetric variances, becomes

better with increasing k. For sufficiently large $k > 10$ the coarse model and the exact model yield comparable accuracy.

3.3.5 Expectation of the Number of Page Accesses

As initially mentioned, the number of page accesses of a nearest neighbor query is equivalent to the number of page accesses of a range query when the nearest neighbor distance is used for the query range. An obvious approach to modeling is therefore to use the expectation of the nearest neighbor distance and to insert it into the expectation of the number of page accesses using range queries:

$$A_{\mathrm{nn}} \approx A_{\mathrm{r}}(R)\,.$$

However, this approach reveals similar statistical problems and leads to similar inaccuracies as the approach in section 3.3.1. The problem is that the number of page accesses is not linear in the query range. Therefore, nearest neighbor distances over the average nearest neighbor distance R are not sufficiently considered. Once again, the approach can be taken if high accuracy is not required or if the variance of the nearest neighbor distance is low.

Instead, we have once again to apply the distribution function $P(r)$ to determine an expectation of the number of page accesses by integration as follows:

$$A_{\mathrm{nn}} = \int_0^\infty A_{\mathrm{range}}(r) \cdot p(r)\partial r\,.$$

For maximum and Euclidean metric, this formula evaluates to:

$$A_{\mathrm{nn,mm}} = \int_0^\infty \left(2r \cdot \sqrt[d]{\frac{N}{C_{\mathrm{eff}}}} + 1 - \frac{1}{C_{\mathrm{eff}}}\right)^d \cdot \left((1-(2r)^d)^{N-1} \cdot (2r)^d\right)\partial r\,,$$

$$A_{\mathrm{nn,em}} = \int_0^\infty \frac{N}{C_{\mathrm{eff}}} \cdot \sum_{0 \le k \le d} \binom{d}{k} \cdot \left(\left(1 - \frac{1}{C_{\mathrm{eff}}}\right) \cdot \sqrt[d]{\frac{C_{\mathrm{eff}}}{N}}\right)^k \cdot \frac{\sqrt{\pi^{d-k}}}{\Gamma\left(\frac{d-k}{2}+1\right)} \cdot r^{d-k}$$

$$\cdot \frac{d \cdot N}{r} \cdot \left(1 - \frac{\sqrt{\pi^d}}{\Gamma(d/2+1)} r^d\right)^N \cdot \frac{\sqrt{\pi^d}}{\Gamma(d/2+1)} r^d \partial r\,.$$

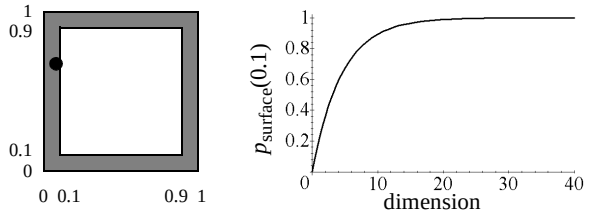


Figure 36: Probability that a Point is Near by the Data Space Boundary.

This result can be simplified by a similar technique as in section 3.3.3.

3.4 Effects in High-Dimensional Data Spaces

In this section, we describe some effects occurring in high-dimensional data space which are not accordingly considered in our models of the previous sections. We still assume a uniform and independent distribution of data and query points in this section. The models developed in the previous sections will be modified to take the described effects into account.

3.4.1 Problems specific to High-Dimensional Data Spaces

The first effect occurring especially in high-dimensional data space is that all data and query points are likely to be near by the boundary of the data space. The probability that a point randomly taken from a uniform and independent distribution in a *d*-dimensional data space has a distance of *r* or below to the space boundary can be determined by the following formula:

$$P_{\text{surface}}(r) \;=\; 1 - (1 - 2 \cdot r)^d .$$

As figure 36 shows, the probability that a point is inside a 10% border of the data space boundary increases rapidly with increasing dimension. It reaches 97% for a 16-dimensional data space.

A second effect which is even more important, is the large extension of query regions. If we use our model for determining an expected value of the nearest neighbor distance, we observe that the expectation approaches fast surprisingly high values. Figure 37

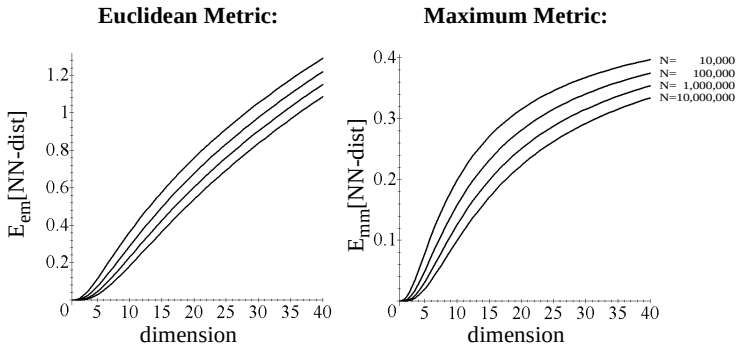


Figure 37: Expected Nearest Neighbor Distance with Varying Dimension.

shows the expected values for the nearest neighbor distance with varying dimension for the maximum metric and the Euclidean metric for several databases containing between 10,000 and 10,000,000 points. Especially using the Euclidean metric, at a data space dimension between 13 and 19, the nearest neighbor distance reaches a value of 0.5, i.e. the nearest neighbor sphere has the same diameter as the complete data space. The size of the database has a small influence on this effect.

The combination of the two effects described above leads us to the observation that large parts of a typical nearest neighbor sphere must exceed the boundary of the data space. The consequences arising from this fact are commonly referred to as *boundary effects*. As we will investigate in depth in the subsequent sections, the most important consequence is that in our models all volume determinations must consider clipping at the boundary of the data space. On the one hand, the expectation of the nearest neighbor distance increases by boundary effects, but on the other hand, access probabilities of data pages decrease because large parts of the Minkowski sum are clipped away.

If dimension further increases, the typical nearest neighbor distance grows to values by far greater than 1/2. In this case, it becomes very likely that the nearest neighbor sphere exceeds most of the data space boundary areas.

A similar effect is observable for the page regions. If we assume, following our initial model, hypercube shaped page regions, the side length of such a region quickly exceeds 0.5. However, it is impossible that the data space is covered only with pages having side lengths between 0.5 and 1. Basically, the pagination arises from a recursive decomposi-

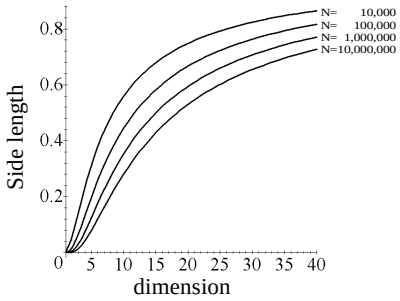


Figure 38: Side Lengths of Page Regions for C_{eff}=30.

tion of the data space into parts of approximately the same volume (for uniformity and independence assumption). Therefore, each page is in each dimension split several times. That means, only the side lengths 1, 1/2, 1/4, 1/8,... (approximately) can occur. In high dimensions, it is simply impossible that a page has a side length of 1/2 or smaller in all dimensions, because if every data page is split at least once in each dimension, we need a total number of at least 2^d data pages to cover the complete data space. For example, in a 30-dimensional data space, we would need one billion data pages to reach such a pagination, resulting in a database size of 4,000 GBytes.

Therefore, we will modify our cost models such that for database sizes N less than $N_{\text{Singlesplit}}$ with

$$N \leq N_{\text{Singlesplit}} = C_{\text{eff}} \cdot 2^d$$

this effect is considered.

3.4.2 Range Query

We still assume uniformity and independence in the distribution of data and query points. For sufficiently high dimension d (such that the inequation above is accomplished), we observe that the data space is only split in a number $d' < d$ of dimensions. The maximum split dimension d' can be determined by using the following formula:

$$d' = \left\lceil \log_2\left(\frac{N}{C_{\text{eff}}}\right) \right\rceil.$$

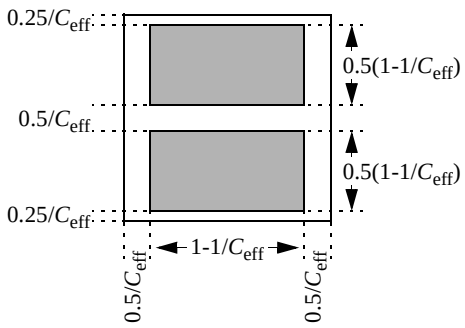


Figure 39: Side Lengths and Positions of Page Regions in the Modified Model.

The data-pages have an average extension a_{split} with

$$a_{split} = 0.5 \cdot \left(1 - \frac{1}{C_{eff}}\right)$$

in d' dimensions and an average extension $a_{unsplit}$ with

$$a_{unsplit} = 1 - \frac{1}{C_{eff}}$$

in the remaining $d' - d$ dimensions. Figure 39 clarifies the position of two typical page regions in the data space for split (y-axis) and unsplit (x-axis) dimensions. The projection on an axis of a split dimension shows 2 page regions. Between these two regions, there is a gap of the average breadth $0.5/C_{eff}$ which is caused by the MBR property of the page region (cf. section 3.2). The distance of $0.25/C_{eff}$ from the data space boundary is also due to the MBR property. In contrast, the projection on an axis of an unsplit dimension shows only one page region with a distance of $0.5/C_{eff}$ from the lower and from the upper space boundary, respectively.

Now, we mark the Minkowski sum of the lower page region (cf. figure 40). We observe that large parts of the Minkowski sum are located outside the data space. The Minkowski sum is the volume, from which a query point has to be taken such that the corresponding page is accessed. On the other hand, we assume that only query points inside the data space boundary are used as query points. Therefore, the Minkowski sum has to be clipped at the data space boundary in order to determine the probability that a

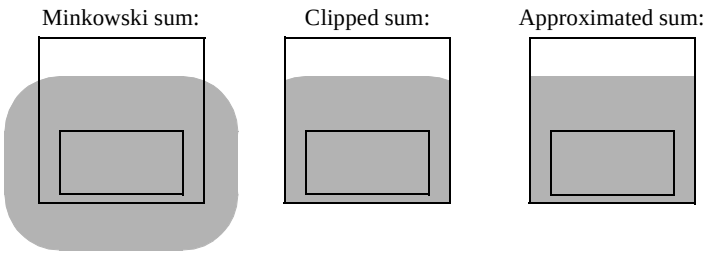


Figure 40: Minkowski Sum Outside the Boundary of the Data Space.

randomly selected query point accesses the page. We can express the clipping on the boundary with the following integral formula which summarizes all points v in the data space (i.e. all possible positions of query points) with a distance less or equal the query range r from the page region R:

$$X_{\mathrm{r,hd,ui}}(r) = V_{(R \oplus S) \cap \mathrm{DS}}(r) = \int_0^1 \cdots \int_0^1 \left(\begin{cases} 1 & \text{if } \delta_M(v, R) \le r \\ 0 & \text{otherwise} \end{cases} \right) \partial v_0 \ldots \partial v_{d-1}.$$

Unfortunately, this integral is difficult to evaluate. Therefore, it has to be simplified. The first observation usable for this purpose is that the distance between the data space boundary and the page region ($0.5/C_{\mathrm{eff}}$ for unsplit dimensions, $0.25/C_{\mathrm{eff}}$ for split dimensions) is small compared to a typical radius r (assuming reasonable selectivities). Therefore, the corresponding gap is always filled, and unsplit dimensions can be ignored for the determination of the access probability (cf. figure 40, right side).

For maximum metric, the clipped Minkowski sum can be determined in the following way (cf. figure 41): We take the side length of the split dimension and fill the small gap

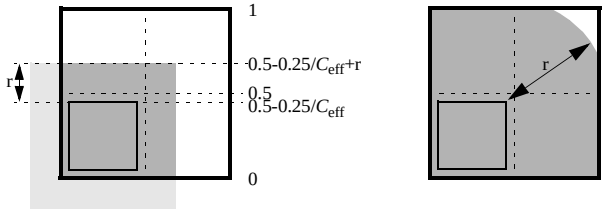


Figure 41: The Modified Minkowski Sum for the Max. (l.) and Euclidean Metric (r.).

to the data space boundary. We add the query radius only one time instead of two times (as we did in our initial model). The result is taken to the power of the number of dimensions split:

$$X_{\mathrm{r,mm,hd,ui}}(r) = \left(\min\left\{0.5 - \frac{0.25}{C_{\mathrm{eff}}} + r, 1\right\}\right)^{d'} = \left(\min\left\{0.5 - \frac{0.25}{C_{\mathrm{eff}}} + r, 1\right\}\right)^{\left\lceil \log_2\left(\frac{N}{C_{\mathrm{eff}}}\right)\right\rceil}.$$

For a radius greater than $0.5 + 0.25 / C_{\mathrm{eff}}$, the Minkowski enlargement reaches also the data space boundary on the opposite. This is taken into account by the application of the minimum function in the equation above. In this case, the page has an access probability of 100%.

If we apply the Euclidean metric, an additional complication arises as depicted on the right side of figure 41. The radius r is typically by far greater than 0.5 (cf. section 3.4.1 and figure 38). Therefore, the spherical part of the Minkowski sum must be clipped. This volume cannot be determined analytically. However, we will show, how this volume can be simplified such that a precomputed volume function can be applied to improve the efficiency of the evaluation.

The basic idea of the precomputation of the clipped sphere volume is to standardize the clipping process to clip only on the unit hypercube. We scale our clipping region such that it is mapped to the unit hypercube. Then, we determine the corresponding volume by looking up in a table of precomputed volumes. After that, we apply the inverse scaling to the volume in the table.

By $V_{\mathrm{csi}}(d,r)$ we define the volume of the intersection of a sphere with radius r and the unit hypercube in the d-dimensional space. We let the origin be the center of the sphere. Obviously, $V_{\mathrm{csi}}(d,0) = 0$ and $V_{\mathrm{csi}}(d, \sqrt[2]{d}) = 1$. Between these points, V_{csi} is monotoni-

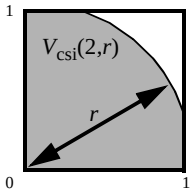


Figure 42: The Volume of the Intersection between Sphere and the Unit Hypercube.

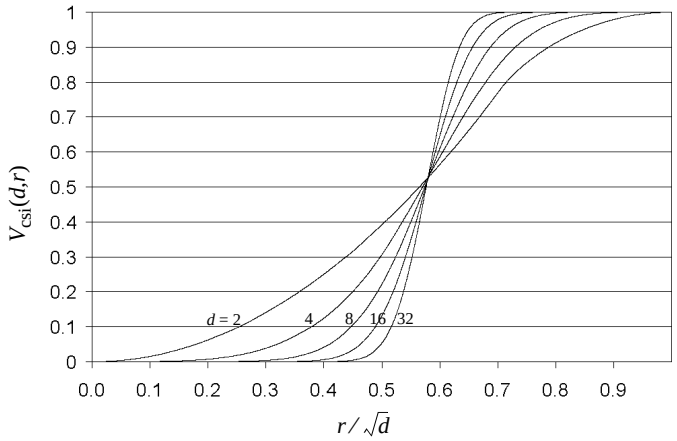


Figure 43: The Volume of the Intersection between a Sphere and the Unit Hypercube.

cally increasing. Figure 42 depicts the intersection volume $V_{csi}(2, r)$ in the 2-dimensional data space.

Definition 7: Cube-Sphere Intersection

$V_{csi}(d,r)$ denotes the intersection volume between the unit hypercube $[0..1]^d$ and a d-dimensional sphere with radius r around the origin:

$$V_{csi}(d, r) = \int_0^1 \ldots \int_0^1 \left(\begin{cases} 1 & \text{if } |P| \le r \\ 0 & \text{otherwise} \end{cases} \right) \partial v_0 \ldots \partial v_{d-1}.$$

$V_{csi}(d,r)$ can be materialized into an array of all relevant dimensions d (e.g. ranging from 1 to 100) and for a discretization of the relevant r between 0 and $\sqrt{d}$ in a sufficiently high number of steps (e.g. 10,000). For the determination of the discretization of $V_{csi}(d,r)$, the Montecarlo integration [Kal 86] using the integral formula can be used. Sufficient accuracy is achievable with 100,000 points.
Figure 43 depicts $V_{csi}(d,r)$ for various dimensions d.

$V_{csi}(d,r)$ is used to determine the access probability of a page when a range query using the Euclidean metric is performed. As we pointed out in the previous discussion, the range query behaves like a range query in the d'-dimensional space, because all

dimensions, where the page region has full extension, can be projected without affecting the volume of the Minkowski enlargement.

However, the query sphere is not clipped by the unit hypercube, but rather by the hypercube representing the empty space between the page region and the opposite boundary of the data space. The side length of this cube a_{empty} is (cf. figures 39-40):

$$a_{\text{empty}} = \frac{1}{2} + \frac{1}{4C_{\text{eff}}}.$$

To determine clipping on such a hypercube, we have to scale the radius accordingly before applying $V_{\text{csi}}(d,r)$:

$$V_{\text{csi}}(d, r, a) = a^d \cdot V_{\text{csi}}(d, \frac{r}{a}).$$

The resulting formula for the access probability using Euclidean range queries is:

$$X_{\text{r,em,hd,ui}}(d', r) = \sum_{0 \le k \le d'} \binom{d'}{k} \cdot \left(\frac{1}{2} - \frac{1}{4C_{\text{eff}}}\right)^k \cdot \left(\frac{1}{2} + \frac{1}{4C_{\text{eff}}}\right)^{d'-k} \cdot V_{\text{csi}}(k, \frac{r}{\frac{1}{2} + \frac{1}{4C_{\text{eff}}}}).$$

To show the impact of the effects in high-dimensional spaces to the estimation of the access probability, figure 44 compares our new function $X_{\text{r,em,hd,ui}}(r)$ with the low-dimensional estimate $X_{\text{r,em,ld,ui}}(r)$ and with an intermediate form, where the volume function V_{csi} is still replaced by the normal sphere volume V_{s}. In the intermediate form only hypersphere segments completely lying outside the data space are clipped. The database contains in this experiment 280,000 points in a 16-dimensional data space. Whereas the

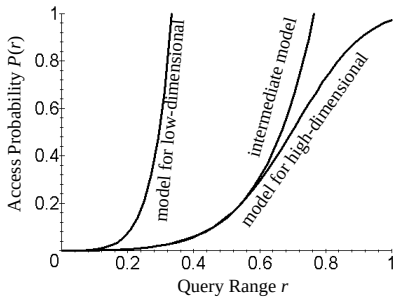


Figure 44: Various Models in High-Dimensional Data Spaces.

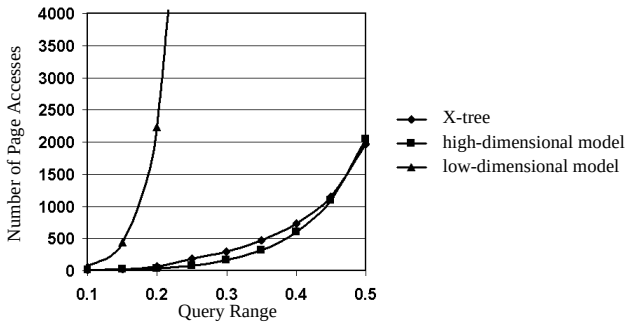


Figure 45: Accuracy of the Models in a 16-dimensional Data Space.

cost model for low-dimensional query processing quickly yields unrealistic access probabilities larger than 100%, the intermediate model is accurate for ranges less than $r = 0.6$. The intermediate model is simpler and more efficient to evaluate, because it does not depend on the precomputed discretization of the volume function V_{csi}.

The expected number of page accesses can be easily estimated if the number of data pages N / C_{eff} is a power of two. In this case, the number of split dimensions is equal for all data pages (although the dimensions, in which the data pages are actually split, may vary). Otherwise, a number of data pages not equal to a power of two requires some data pages to be split once more than the rest. As the number of all data pages is $n_{\text{dp}} = N / C_{\text{eff}}$, the number of data pages split once more than the others $n_{d'}$ is equal to:

$$n_{d'} = 2 \cdot \left(\frac{N}{C_{\text{eff}}} - 2^{\left\lfloor \log_2(\frac{N}{C_{\text{eff}}}) \right\rfloor} \right).$$

Likewise, the number of data pages split one time fewer $n_{d'-1}$ is equal to:

$$n_{d'-1} = n_{\text{dp}} - n_{d'}.$$

Then, the expected number of page accesses is equal to:

$$A_{\text{hd}}(r) = n_{d'} \cdot X_{\text{hd}}(\left\lceil \log_2(\frac{N}{C_{\text{eff}}}) \right\rceil, r) + n_{d'-1} \cdot X_{\text{hd}}(\left\lfloor \log_2(\frac{N}{C_{\text{eff}}}) \right\rfloor, r).$$

This equation holds for maximum metric as well as Euclidean metric and for range queries as well as nearest neighbor queries.

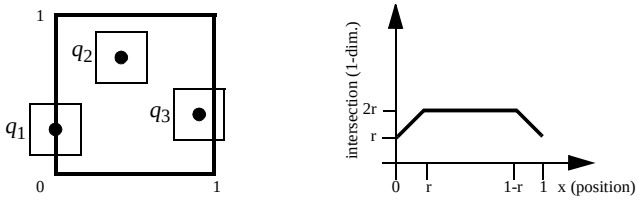


Figure 46: The Intersection Volume for Maximum Metric and Arbitrary Center Point.

The accuracy of the low-dimensional and the high-dimensional cost model for range query processing was compared by using a database of 100,000 points taken from a uniform, independent data distribution in the 16-dimensional data space. The query range was varied from 0.1 to 0.5 using the maximum metric, yielding selectivities between $6.5 \cdot 10^{-12}$ and 11.8%. The results are depicted in figure 45. As expected, the high-dimensional model yields a reasonable accuracy, whereas the low-dimensional model completely fails in this case.

3.4.3 Nearest Neighbor Query

Typically, query spheres exceed the data space boundary in high-dimensional query processing. For range queries, the consequence is a smaller result set compared with the expectation when neglecting this boundary effect, because only the part of the sphere inside the data space is able to contribute to the result set. In contrast, nearest neighbor queries have a fixed result set size (1 point for a 1-nearest neighbor query). The consequence here is that a greater radius is needed to achieve the same result set size in the presence of boundary effects. The nearest neighbor distance is increased by boundary effects.

First, we develop an expectation for the volume $V_{csi,a}(d,r)$ of the intersection volume of the unit hypercube and a sphere with radius r, whose center is arbitrarily chosen in the unit hypercube. We note that this task is similar to the intersection volume $V_{csi}(d,r)$ in the previous subsection. However, the center of the sphere is now arbitrarily chosen and not fixed in the origin. $V_{csi,a}(d,r)$ corresponds to the probability that two points arbitrarily chosen from the unit hypercube have a distance less or equal to r from each other.

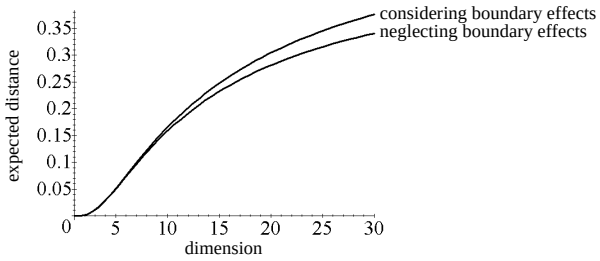


Figure 47: The Impact of Boundary Effects on the Nearest Neighbor Distance.

When the maximum metric is used for the query, the expectation for the intersection volume, which is an intersection of two hypercubes, can be determined analytically. Figure 46 depicts three different positions of queries in the data space. First, we consider only the projection on the *x*-axis. The center point of q_1 lies exactly on the lower space boundary. Therefore, only half of the side length (r) is inside the data space. The center point of q_2 has a distance greater than r from the data space boundary. Therefore, the complete side length of the cube ($2r$) is inside the data space. Query q_3 intersects the right space boundary, but more than half of the side length is inside the data space. The right diagram of figure 46 depicts the progression of the part of the side length which is inside the data space with varying position of the query point. It is r at the points 0 and 1, $2r$ between the positions r and $1-r$. Between 0 and r, the intersection increases linearly. The average of the intersection over all positions is:

$$V_{\text{cci,a}}(1, r) = 2 \cdot r - \frac{r^2}{2}.$$

We can extend this result to the d-dimensional case simply by taking the power of d:

$$V_{\text{cci,a}}(d, r) = V_{\text{cci,a}}(1, r)^d = \left(2 \cdot r - \frac{r^2}{2}\right)^d.$$

This result can be used to determine the expectation of the nearest neighbor distance. A completely analytical solution is possible if we apply our coarse estimation by equalizing $V_{\text{cci,a}}(d,r)$ with $1/N$:

$$\frac{1}{N} = V_{\text{cci,a}}(d, r) = \left(2 \cdot r - \frac{r^2}{2}\right)^d \qquad r = 2 - \sqrt{4 - 2 \cdot \sqrt[d]{\frac{1}{N}}}.$$

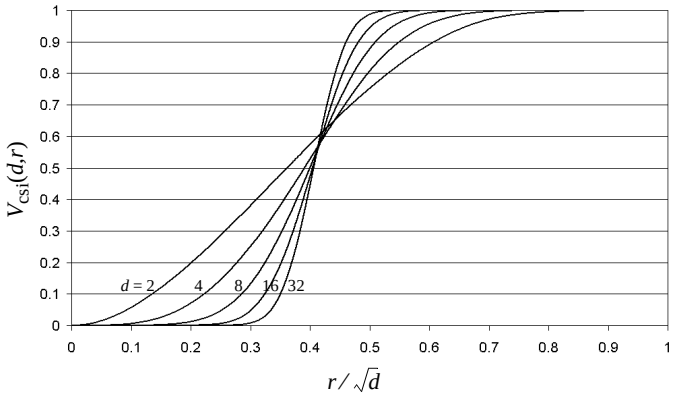


Figure 48: The Intersection Volume for Euclidean Metric and Arbitrary Center Point.

The impact of boundary effects on the nearest neighbor distance is shown in figure 47. As expected, boundary effects do not play an important role in low dimensions up to 10. With increasing dimension, the effect becomes more important. Neglecting boundary effects, we underestimate the nearest neighbor distance by 10% in the 30-dimensional space.

The new volume determination $V_{cci,a}(d,r)$ can also be applied in our exact model for nearest neighbor estimation. The corresponding probability distribution is in this case:

$$P_{mm,hd}(r) = 1 - (1 - V_{cci,a}(d, r))^N = 1 - \left(1 - \left(2 \cdot r - \frac{r^2}{2}\right)^d\right)^N .$$

The probability density $p_{mm,hd}(r)$, the expectation for the nearest neighbor distance $R_{mm,hd}$, and the expectation of the number of page accesses $A_{nn,mm,hd}$ can be derived from the probability distribution as described in section 3.3.

When Euclidean metric is applied, the same problem arises as in section 3.4.2. It is difficult to determine the intersection volume between the unit hypercube and a sphere with arbitrary center in an analytical way. To cope with this problem, a similar precomputation of the volume may be used. Again, we define the *Cube-Sphere Intersection with Arbitrary Center*, $V_{csi,a}(d,r)$ by a multidimensional integral which can be evaluated by using the Montecarlo integration [Kal 86]. The result can be stored in an array for use by the model.

Definition 8: Cube-Sphere Intersection with Arbitrary Center

$V_{csi,a}(d,r)$ denotes the intersection volume between the unit hypercube $[0..1]^d$ and a d-dimensional sphere with radius r around a point arbitrarily chosen from the unit hypercube:

$$V_{csi,a}(d, r) = \int_{(0,\ldots,0)}^{(1,\ldots,1)} \left(\int_{(0,\ldots,0)}^{(1,\ldots,1)} \left(\begin{cases} 1 & \text{if } |v - w| \le r \\ 0 & \text{otherwise} \end{cases} \right) \partial v \right) \partial w .$$

Figure 48 shows $V_{csi,a}(d,r)$ for the dimensions 2, 4, 8, 16 and 32. The intersection volume was determined for all dimensions between 1 and 100 for each radius between 0 and $\sqrt{d}$ in 10,000 intervals using 100,000 steps of the Montecarlo integration [Kal 86]. The 10,000 intervals can be used for an efficient numerical evaluation of the expectation:

The probability distribution of the nearest neighbor distance r considering boundary effects is for the Euclidean metric:

$$P_{em,hd}(r) = 1 - (1 - V_{csi,a}(d, r))^N .$$

The corresponding density function is:

$$p_{em,hd}(r) = \frac{\partial P_{em,hd}(r)}{\partial r} = N \cdot (1 - V_{csi,a}(d, r))^{N-1} \cdot \frac{\partial V_{csi,a}(d, r)}{\partial r} .$$

Provided that $V_{csi,a}[d,i]$ is an array with the range $[1..d_{max},0..i_{max}]$ which contains the precomputed values of $V_{csi,a}(d,r)$ for r ranging from 0 to $\sqrt{d}$ with

$$i = \left\lfloor \frac{r \cdot i_{max}}{\sqrt{d}} \right\rfloor ,$$

we are able to replace integral formulas such as the expected value of the nearest neighbor distance by a finite summation:

$$R_{em,hd} = \int_0^{\infty} r \cdot p_{em,hd}(r) \partial r$$

$$= N \cdot \int_0^{\sqrt{d}} r \cdot (1 - V_{csi,a}(d, r))^{N-1} \cdot \frac{\partial V_{csi,a}(d, r)}{\partial r} \partial r$$

$$\approx N \cdot \sum_{i=1}^{i_{max}} \frac{i \cdot d}{i_{max}^2} \cdot (1 - V_{csi,a}[d, i])^{N-1} \cdot \frac{V_{csi,a}[d, i] - V_{csi,a}[d, i-1]}{(\sqrt{d} / i_{max})}$$

$$= \frac{N \cdot \sqrt{d}}{i_{max}} \cdot \sum_{i=1}^{i_{max}} i \cdot (1 - V_{csi,a}[d, i])^{N-1} \cdot (V_{csi,a}[d, i] - V_{csi,a}[d, i-1]).$$

The infinite upper bound of the integral can be replaced by $\sqrt{d}$, because the derivative of $V_{csi,a}(d,r)$ is constantly 0 for r larger than $\sqrt{d}$, while all other terms have finite values. The derivative is replaced by the local difference.

The expected value of the number of page accesses can be determined in the same way:

$$A_{nn,em,hd} = \int_0^{\infty} A_{r,em,hd}(r) \cdot p_{em,hd}(r) \partial r$$

$$= N \cdot \sum_{i=1}^{i_{max}} A_{r,em,hd}\left(\frac{i\sqrt{d}}{i_{max}}\right) \cdot (1 - V_{csi,a}[d, i])^{N-1} (V_{csi,a}[d, i] - V_{csi,a}[d, i-1]) \ .$$

The evaluation of these formulas is efficient, because the required volume functions $V_{csi,a}(d,r)$ and $V_{csi}(d,r)$ are independent of any database specific setting such as the number of points in the database, the point density or the effective page capacity C_{eff}. The predetermined discretization of these functions requires a few megabytes of storage and can be statically linked with the programs evaluating cost models. Costly Montecarlo integration processes are run only at compile time, not at run-time. Further improvement is achievable if we consider that the probability density only contributes in the interval between r_{lb} and r_{ub} (cf. section 3.3.3). Integration and summation can be bounded to this area:

$$R_{em,hd} \approx N \cdot \sum_{i=\left\lfloor \frac{r_{lb} \cdot i_{max}}{\sqrt{d}} \right\rfloor}^{\left\lceil \frac{r_{ub} \cdot i_{max}}{\sqrt{d}} \right\rceil} \frac{i \cdot \sqrt{d}}{i_{max}} \cdot (1 - V_{csi,a}[d, i])^{N-1} \cdot \frac{V_{csi,a}[d, i] - V_{csi,a}[d, i-1]}{\sqrt{d} / i_{max}} \ .$$

To evaluate the cost formula for query processing using nearest neighbor queries, we constructed indexes with varying data space dimensionality. All databases contained

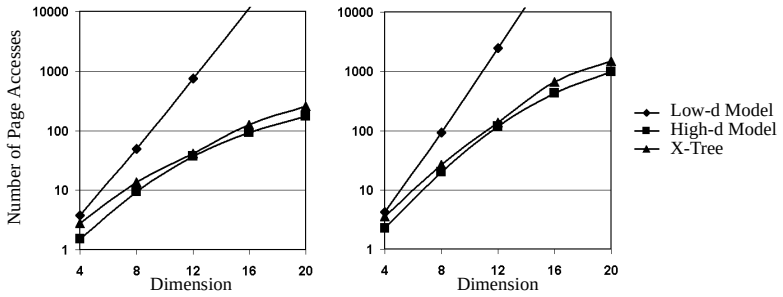


Figure 49: Accuracy of the Cost Models for Nearest Neighbor Queries.

100,000 points taken from a uniform and independent distribution. The effective capacity of the data pages was 48.8 in all experiments (the block-size was chosen correspondingly). The dimension varied from 4 to 20. We performed nearest neighbor queries using maximum metric and Euclidean metric on all these indexes and compared the observed page accesses with the predictions of the low-dimensional and the high-dimensional model developed in this chapter. The results are depicted in figure 49. The diagram on the left side shows the results for maximum metric, the right diagram shows the results for Euclidean Metric. Whereas the cost model for high-dimensional query processing provides accurate estimates over all dimensions, the low-dimensional model is only accurate in the low-dimensional area up to $d = 8$. Beyond this area, the low-dimensional model completely fails to predict the number of page accesses. Not even the order of magnitude is correctly revealed by the low-dimensional model. We should note that the low-dimensional model is mainly related to the original model of Friedman, Bentley and Finkel [FBF 77] and the extension of Cleary [Cle 79].

3.5 Data Sets from Real-World-Applications

It has been extensively investigated that data sets from real applications consistently violate the assumptions of uniformity and independence [FK 94, BF 95]. In this section, we describe the effects and adapt our models to take non-uniformity and correlation into account.

3.5.1 Independent Non-Uniformity

It was already proven in the well-known cost model by Friedman, Bentley and Finkel [FBF 77] that non-uniformity has no influence on the cost of nearest neighbor query processing if no correlation occurs and if the data distribution is smooth. Smoothness means in this context that the point density does not vary severely inside the Minkowski enlargement of a page region. The intuitive reason is the following: Query points are assumed to be taken from the same distribution as data points. For the access probability of a page, we have to determine the fraction of query points which are inside the Minkowski enlargement of the page. If the point density is over the average in some region (say by a factor c) due to non-uniformity, then both, the average volume of the page regions and the average volume of the query regions are scaled by the factor $1/c$. This means that the Minkowski sum is scaled by $1/c$. But then, the number of points inside a given volume is by a factor of c higher than in the uniform case. Therefore, the number of points in the Minkowski enlargement is the same as in the uniform case.

Range queries are difficult to model in the case of non-uniformity, because in the same way as the point density changes with varying location, the size of the result set and the number of page accesses will change.

3.5.2 Correlation

For real data both the assumption of independent non-uniform data distribution is as unrealistic as the assumption of independent uniform distribution. One of the most important properties of real data is the correlation.

Correlation means that one or more attribute values are dependent on the values of one or more other attributes. Typically, the dependence is not strict in the sense that the depending value can be directly determined from the other attributes. We can observe a small interval or a small set of possible values where the depending attribute is located with high probability.

The geometrical meaning of a correlation is the following: The d-dimensional space is not completely covered with data points. Instead, all points are collected on a lower-dimensional area which is embedded in the data space. An example is shown in figure 50, where all data points are located on a 1-dimensional line which is embedded in the 2-dimensional data space. As depicted, the line is not necessarily a straight line. It is also possible that there are several lines which are not connected, or that the data points are located in a cloud around the line.

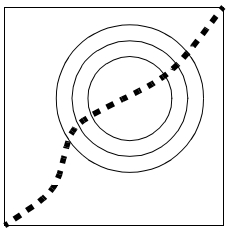


Figure 50: Correlations and their Problems.

A concept which is often used to handle correlation is the singular value decomposition (*SVD*) [DH 73, Fuk 90, GL 89] or the principal component analysis (*PCA*) [FL 95, PFTV 88, Str 80]. These techniques transform the data points directly in a lower-dimensional data space by rotation operations and eliminate the correlation in this way. The point set is indexed in the lower-dimensional space, and query processing can be modeled by using our techniques presented in section 3.2 - 3.5.

However, SVD and PCA can only detect and eliminate linear correlations. A linear correlation means a single straight line in our example. If there are, for example, two warped lines where data points are located on, SVD and PCA will completely fail. We will show later that the performance of query processing anyway takes benefit from the fact that the actual dimension of the point set is lower than the dimension of the data space. Therefore, an explicit transformation is not required.

The general problem of correlations is also depicted in figure 50. If we blow up a circle around some data point, we observe that the number of points is not proportional to the area of the circle as we would expect. Because the actual dimension of the data set is 1, the number of points enclosed in a circle with radius r is proportional to the radius r. The same observation is valid if we blow up a cube or some other d-dimensional object which does not prefer single dimensions for the extension.

This provides us with a means to define the actual dimension of the data set. Under uniformity and independence assumptions the number of points enclosed in a hypercube with side length s is proportional to the volume of the hypercube:

$$n_{\text{encl}} = \rho \cdot s^d = \rho \cdot V.$$

Real data sets form a similar power law using the fractal dimension D_F of the data set:

$$n_{\text{encl}} = \rho_F \cdot s^{D_F} = \rho_F \cdot V^{D_F/d}$$

where ρ_F is the fractal analogue to the point density ρ. The power law was used by Faloutsos and Kamel [FK 94] for the 'box counting' fractal dimension. We use this formula directly for the definition of the fractal dimension:

Definition 9: Fractal Dimension

The fractal dimension D_F of a point set is the exponent which the following power law is valid for:

$$n_{\text{encl}} = \rho_F \cdot V^{D_F/d}.$$

Basically, the fractal dimension is not constant over all scales. It is possible that the fractal dimension changes, depending on the size of the volume *V*. In practice, the fractal dimension is often constant over the wide range of the relevant scales. It is also possible that the fractal dimension is not location-invariant, i.e. a subset of the data set forms a different fractal dimension than the rest of the data set. Intuitively, a reason for this behavior can be that our database contains different kinds of objects (e.g. oil paintings and photos in an image database).

3.5.3 Model Dependence on the Fractal Dimension

Our first consequence to the observation of a fractal dimension D_F is that it is dependent on D_F rather than on the embedding dimension d which model is to use. If D_F is small, then most data points and most queries are far away from the data space boundary. Therefore, we need not apply clipping on the data space boundary and we must not consider clipping in our model. For this case, we have to adapt the cost model for low-dimensional data spaces (cf. section 3.2-3.3). In contrast, if D_F is large, effects of high-dimensional data spaces occur. Therefore, the model for high-dimensional data spaces must be adapted for this case (cf. section 3.4). For moderate D_F both basic models can be applied. For reasons of practicability, we assume boundary effects if the fractal dimension D_F is greater or equal to the maximum split dimension d':

$$D_F \geq d' = \left\lceil \log_2\left(\frac{N}{C_{\text{eff}}}\right) \right\rceil.$$

3.5.4 Range Query

First, we want to determine, how the access probability of a page changes in presence of a correlation described by the fractal dimension D_F. Let us assume that the fractal point density ρ_F is constant throughout the area of the page region and its Minkowski enlargement. In the case of low D_F, we can estimate the side length a of a page region according to the power law:

$$C_{\text{eff}} = \rho_F \cdot \left(\frac{a_{\text{ld}}}{1 - \frac{1}{C_{eff}}} \right)^{D_F} ; a_{\text{ld}} = \sqrt[D_F]{\frac{C_{\text{eff}}}{\rho_F}} \cdot \left(1 - \frac{1}{C_{eff}}\right).$$

In the high-dimensional case, we have still d' splits explicitly applied to achieve data pages with a suitable number of points (C_{eff}). However, we must take into account that a split in some dimension automatically leads to a reduced extension in some correlated dimension. We assume the extension

$$a_{\text{split}} = 0.5 \cdot \left(1 - \frac{1}{C_{\text{eff}}}\right)$$

(cf. section 3.4.2) in a number d'' of dimensions with

$$d'' = \frac{d' \cdot d}{D_F}$$

and full extension (up to MBR effects, cf. section 3.4.2)

$$a_{\text{unsplit}} = 1 - 1/C_{\text{eff}}$$

in the remaining $d - d''$ dimensions.

The Minkowski sum of the page region and a query range r corresponds to the access probability of the page under the assumptions that data points are correlated and that query points are taken from a uniform, independent distribution. Following our discussion in section 3.4.2, we get the following access probabilities for Euclidean metric and maximum metric and for the high-dimensional and the low-dimensional case, respectively:

$$X_{\text{r,mm,ld,c/ui}}(r) = \left(\min \left\{ \sqrt[D_F]{\frac{C_{\text{eff}}}{\rho_F}} \cdot \left(1 - \frac{1}{C_{eff}}\right) + r, 1 \right\} \right)^d ,$$

$$X_{\mathrm{r,mm,hd,c/ui}}(r) = \left(\min\left\{0.5 - \frac{0.25}{C_{\mathrm{eff}}} + r,\, 1\right\}\right)^{\left\lceil \log_2\left(\frac{N}{C_{\mathrm{eff}}}\right)\right\rceil \cdot \frac{d}{D_F}},$$

$$X_{\mathrm{r,em,ld,c/ui}}(r) = \sum_{0 \le k \le d} \binom{d}{k} \cdot \left(\left(1 - \frac{1}{C_{\mathrm{eff}}}\right) \cdot \sqrt[D_F]{\frac{C_{\mathrm{eff}}}{N}}\right)^k \cdot \frac{\sqrt{\pi^{d-k}}}{\Gamma\left(\frac{d-k}{2} + 1\right)} \cdot r^{d-k},$$

$$X_{\mathrm{r,em,hd,c/ui}}(r) = \sum_{0 \le k \le d''} \binom{d''}{k} \cdot \left(\frac{1}{2} - \frac{0.25}{C_{\mathrm{eff}}}\right)^k \cdot \left(\frac{1}{2} + \frac{0.25}{C_{\mathrm{eff}}}\right)^{d''-k} \cdot V_{\mathrm{csi}}(d'', \frac{r}{\frac{1}{2} + \frac{0.25}{C_{\mathrm{eff}}}}).$$

Starting from this point, the expectation for the number of page accesses can be easily determined by multiplication with the number of pages N / C_{eff}.

For real applications, uniform distribution of the query points is not a realistic assumption. A better alternative is to assume that data points and query points are taken from the same distribution and yield the same fractal dimension D_F. Instead of taking the volume of the Minkowski enlargement for the access probability, we should rather determine the percentage of the query points lying in the Minkowski enlargement. The power law can be used for this purpose, such as:

$$\begin{aligned} X_{\mathrm{r,mm,ld,c}}(r) &= X_{\mathrm{r,mm,ld,c/ui}}(r)^{D_F/d} \\ &= \left(\min\left\{\sqrt[D_F]{\frac{C_{\mathrm{eff}}}{\rho_F}} \cdot \left(1 - \frac{1}{C_{eff}}\right) + r,\, 1\right\}\right)^{D_F}. \end{aligned}$$

The other equations can be modified in the same way.

3.5.5 Nearest Neighbor Query

Following our coarse model for the estimation of the nearest neighbor distance (cf. section 3.3.1), we can easily determine a volume having an expectation of 1 point enclosed. Like in the preceding section, we assume that the distribution of the query points follows the distribution of the data points. The volume can then be estimated by using the power law:

$$1 = \rho_F \cdot V^{D_F/d} \qquad R_{\mathrm{mm,ld,cor}} \approx \frac{1}{2 \cdot \sqrt[D_F]{\rho_F}}$$

for the maximum metric and

$$\frac{1}{\rho_F} = V^{D_F/d} = \frac{\sqrt{\pi}^{D_F}}{\Gamma(d/2+1)^{D_F/d}} \cdot r^{D_F} \qquad R_{\text{em,ld,cor}} \approx \frac{\sqrt[d]{\Gamma(d/2+1)}}{\sqrt{\pi}} \cdot \sqrt[D_F]{\frac{1}{\rho_F}}$$

for the Euclidean metric. If D_F is sufficiently large (according to section 3.5.4), boundary effects must be considered. For the maximum metric, we get the following formula:

$$1 = \rho_F \cdot \left(2r - \frac{r^2}{2}\right)^{D_F} \qquad R_{\text{mm,hd,cor}} \approx 2 - \sqrt{4 - 2 \cdot \sqrt[D_F]{\frac{1}{\rho_F}}}.$$

For the Euclidean metric, we need the inverse function of the *cube-sphere intersection with arbitrary center*, $V^{-1}_{\text{csi,a}}(d, r)$ (cf. section 3.4.3). The corresponding discretization of $V^{-1}_{\text{csi,a}}(d, r)$ can be gained in a single pass of the discretization of $V_{\text{csi,a}}(d, r)$. The estimation of the nearest neighbor distance is:

$$\frac{1}{\rho_F} = V_{\text{csi,a}}(r)^{D_F/d} \qquad R_{\text{em,hd,cor}} \approx V^{-1}_{\text{csi,a}}\left(\left(\frac{1}{\rho_F}\right)^{d/D_F}\right).$$

For our exact model, we have to adapt our distribution function in a suitable way. Again, we have to apply the power law:

$$P(r) = 1 - \left(1 - \frac{\rho_F}{N} \cdot V(r)^{\frac{D_F}{d}}\right)^N,$$

where $V(r)$ is the volume of the d-dimensional hypersphere with radius r in the case of the Euclidean metric and the volume of the d-dimensional hypercube with side length $2r$ in the case of the maximum metric. We have to make a suitable distinction between the low-dimensional and the high-dimensional case when choosing $V(r)$. The rest is straightforward and can be handled as in section 3.3-3.4: An expectation for the nearest neighbor distance can again be gained by integrating r multiplied with the derivative of $P(r)$. The new distribution function must be multiplied with the Minkowski sum as in section 3.3-3.4. For the maximum metric, we get the following formulas for the low-dimensional and the high-dimensional case, respectively:

$$A_{\text{nn,mm,ld,c}} = \frac{N}{C_{\text{eff}}} \int_0^\infty \left(\left(\min\left\{\sqrt[D_F]{\frac{C_{\text{eff}}}{\rho_F}} \cdot \left(1 - \frac{1}{C_{eff}}\right) + r, 1\right\}\right)^{D_F} \cdot \frac{\partial}{\partial r}\left(1 - \left(1 - \frac{\rho_F}{N} \cdot (2r)^{D_F}\right)^N\right)\right) \partial r,$$

$$A_{\text{nn,mm,hd,c}} = \frac{N}{C_{\text{eff}}} \int_0^{\infty} \left(\left(\min \left\{ \left(\frac{1}{2} - \frac{0.25}{C_{eff}} \right) + r, 1 \right\} \right)^{\left\lceil \log_2\left(\frac{N}{C_{\text{eff}}}\right) \right\rceil} \cdot \frac{\partial}{\partial r} \left(1 - \left(1 - \frac{\rho_F}{N} \cdot \left(2r - \frac{r^2}{2} \right)^{D_F} \right)^N \right) \right) \partial r .$$

For the Euclidean metric, the corresponding result is

$$A_{\text{nn,em,ld,c}} = \frac{N}{C_{\text{eff}}} \int_0^{\infty} \left(\left(\sum_{0 \le k \le d} \binom{d}{k} \cdot \left(\left(1 - \frac{1}{C_{\text{eff}}} \right) \cdot \sqrt[D_F]{\frac{C_{\text{eff}}}{N}} \right)^k \cdot \frac{\sqrt{\pi}^{d-k}}{\Gamma\left(\frac{d-k}{2} + 1 \right)} \cdot r^{d-k} \right)^{\frac{D_F}{d}} \right.$$

$$\left. \cdot \frac{\partial}{\partial r} \left(1 - \left(1 - \frac{\rho_F}{N} \cdot \frac{\sqrt{\pi}^{D_F}}{\Gamma(d/2+1)^{D_F/d}} \cdot r^{D_F} \right)^N \right) \right) \partial r ,$$

$$A_{\text{nn,em,hd,c}} = \frac{N}{C_{\text{eff}}} \int_0^{\infty} \left(\left(\sum_{0 \le k \le d''} \binom{d''}{k} \cdot \left(\frac{1}{2} - \frac{0.25}{C_{\text{eff}}} \right)^k \cdot \left(\frac{1}{2} + \frac{0.25}{C_{\text{eff}}} \right)^{d''-k} \cdot V_{\text{csi}}(d'', \frac{r}{\frac{1}{2} + \frac{0.25}{C_{\text{eff}}}}) \right)^{\frac{D_F}{d}} \right.$$

$$\left. \cdot \frac{\partial}{\partial r} \left(1 - \left(1 - \frac{\rho_F}{N} \cdot V_{\text{csi,a}}(d, r)^{\frac{D_F}{d}} \right)^N \right) \right) \partial r .$$

Rules facilitating the evaluation of these formulas were presented in section 3.3-3.4.

To evaluate our model extension, indexes on several data sets from real-world applications (cf. chapter 1) were constructed. Our first application is a similarity search system for CAD drawings provided by a subcontractor of the automobile industry [BK 97]. The drawings were transformed into 16-dimensional fourier vectors. Our second application is a content based image retrieval using color histograms with 16 histogram bins [Sei 97]. Both databases contained 50,000 objects. Our third application contained 9-dimensional vectors from weather observations. The fractal dimensions of our data sets are 7.0 (CAD), 8.5 (Color Histograms) and 7.3 (Clouds). We performed nearest neighbor queries using the Euclidean and the maximum metric and compared the results obtained with the predictions of the following 3 cost models:

- The original models by Friedman, Bentley and Finkel [FBF 77] and Cleary [Cle 79], cf. section 3.3
- our extension to high-dimensional query processing (cf. section 3.4)

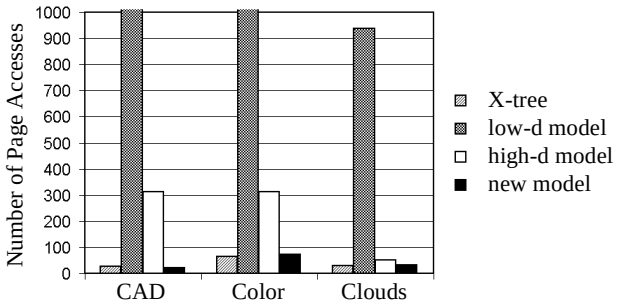


Figure 51: Accuracy for Data Sets from Real-World Applications.

- our extension to non-uniformity and correlation

The results are depicted in figure 51. In contrast to the low-dimensional and the high-dimensional model, the new model considering correlation yields sufficient accuracy in all performed experiments.

3.6 Modeling the Storage System

The physical structure of a disk drive [PH 90, Sie 90, SPG 91] is depicted in figure 52: The disk drive consists of a row of magnetic disks which are fixed above each other on a common axis. The disks rotate with high speed. Data stored on the disk is accessed by a set of disk heads fixed on a common disk arm which can be moved orthogonally to the rotation direction (i.e. in radial direction).

If some data is requested from a random position on the disk drive, the following single actions are performed: First, the disk head is moved to the corresponding track (*positioning time*). Then, the system waits until the requested data is positioned below the disk head (*rotational delay time*). Finally, the data is transferred to the main memory (*transfer time*). The unit of transfer between the disk drive and the main memory is a *sector* or *physical page*. There is no positioning delay if the disk head is already positioned at the right track of the right disk. This case happens if contiguous blocks are read in separate reading actions from disk. If the access is switched from one magnetic disk to a different one, the disk heads have to be repositioned, because the tracks of different disks are not exactly aligned.

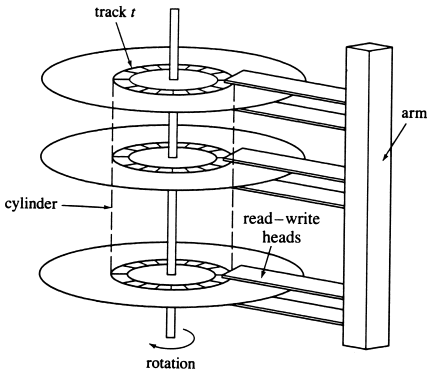


Figure 52: Structure of a Disk Drive [SPG 91].

For indexes which are dynamically constructed and which do not consider relative positions of pages on disks in their construction algorithms and in their query processing algorithms (such as all structures presented in chapter 2), we can make the following simplifying assumptions:

All accesses have a constant logical blocksize which is not identical to the physical page size. According to the author's experience, it is not even important that logical blocks are correctly aligned to physical blocks, because the additional time for contiguously reading a superfluous sector is negligible compared to *positioning time* and *delay time*.

Every access is independent of the preceding access. Therefore, the disk arm is repositioned for almost every access. We summarize the *positioning time* and the *rotational delay time* to the *seek time*. To determine typical values for *seek time* and *transfer time* for nowaday's disk drives, we performed the following experiment: We measured block accesses with varying block size and random position in a large file. The result is presented in figure 53 (access time plus/minus standard deviation). Obviously, the access time is perfectly linear in the logical block size. We observe the following law for the access time t_{acc}:

$$t_{access} = t_{seek} + b \cdot t_{transfer},$$

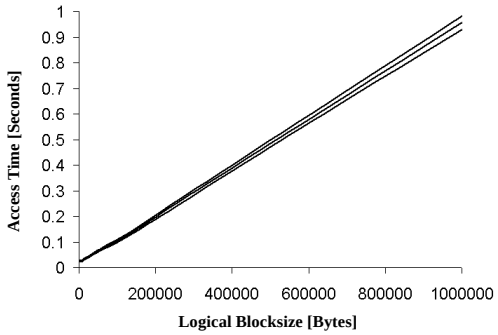


Figure 53: Access Time of Disk Drive with Varying Logical Blocksize.

where b is the size of the block in bytes.

We can determine seek time and transfer time by linear regression. In our example, we get the following values:

$$t_{seek} = 20 \text{ msec},$$
$$t_{transfer} = 975 \text{ nsec/Byte}.$$

In these values, the overhead of the file system and the basic load of disk drives in a typical UNIX system is considered. Neglecting the basic load leads to a seek time of 12 msec and a transfer time of 200 nsec/Byte.

Chapter 4
Dynamic Optimization of the Logical Block Size

The first application of our cost model presented in chapter 3 is the optimization of the logical block size used in the index. For this purpose, we propose a special new index structure which is capable of adapting the logical block size dynamically and independently in different pages of the index.

4.1 Motivation

In recent years, a general criticism on high-dimensional indexing has come up. Most multidimensional index structures have an exponential dependency (with respect to the time for processing range queries and nearest neighbor queries) upon the number of dimensions. To illustrate this, figure 54 shows our model prediction of the processing time of the X-tree for a uniform and independent data distribution (constant database size 400 KBytes). With increasing dimension d, the processing time grows exponentially until saturation comes into effect, i.e. a substantial ratio of all index pages is accessed. In very high dimensions $d \geq 25$, virtually all pages are accessed, and the processing time approaches thus an upper limit.

In recognition of this fact, an alternative approach is simply to perform a sequential scan over the entire data set. The sequential scan causes substantially fewer effort than

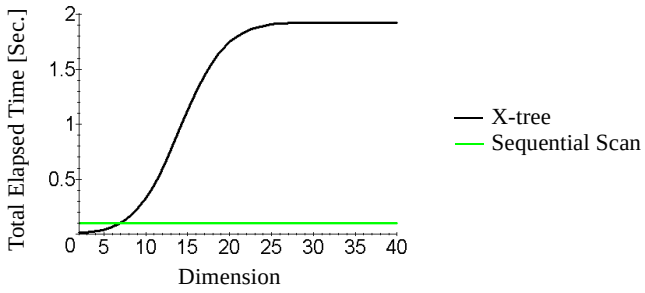


Figure 54: Performance of Query Processing With Varying Dimension.

processing all pages of an index, because the reading operations in the index cause random seek operations whereas the scan reads sequentially. The sequential scan causes seldom disk arm movements or rotational delays which are of no consequence compared to the transfer cost. Assuming a logical block size of 4 KBytes, contiguous reading of a large file is by a factor >12 faster than reading the same amount of data from random positions (cf. section 3.6).

A second advantage of the sequential scan over index-based query processing is its storage utilization of 100%. In contrast, index pages have a storage utilization between 60% and 70% which causes a further performance advantage of about 50% for the sequential scan when reading the same amount of data. The constant cost of the sequential scan is also depicted in figure 54. The third advantage of the sequential scan is the lacking overhead of processing the directory. We can summarize that the index must not access more than 5% of the pages in order to remain competitive with the sequential scan.

In figure 54, the break-even point of the two techniques is reached at $d = 7$. The trade-off between the two techniques, however, is not simply expressed in terms of the number of dimensions. For instance when data sets are highly skewed (as real data sets often are), index techniques remain more efficient than a scan up to a fairly high dimension. Similarly, when there are correlations between dimensions, index techniques tend to benefit compared with scanning. Obviously, the number of data objects currently stored in the database plays an important role since the sequential scan is linear in the number of objects whereas query processing based on indexes is sub-linear.

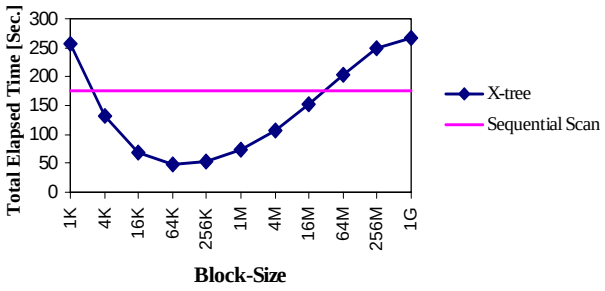


Figure 55: Block Size Optimization.

Figure 55 shows the model predictions of the X-tree for 10,000,000 points uniformly and independently chosen from a 20-dimensional data space with varying block size from 1 KByte to 1 GByte. In this setting, the performance is relatively bad for usual block sizes between 1 KBytes and 4 KBytes, fast improving when increasing the block size. A broad and stable optimum is reached between 64 KBytes and 256 KBytes. Beyond this optimum, the performance deteriorates again. Due to the storage utilization below 100%, the sequential scan outperforms the X-tree for very large block sizes. This result shows that block size optimization is the most important advice to improve high-dimensional indexes.

The rest of this chapter is organized as follows: Section 4.2 explains the general idea and an overview of our technique. Section 4.3 shows the architectural structure of the DABS-tree. The following sections show how operations such as insert, deletion and search are handled. In section 4.7 we show how our model developed in chapter 3 can be applied for a dynamic and independent optimization of the logical block size. Finally, we present an experimental evaluation of our technique.

4.2 Basic Idea

As we pointed out in section 4.1, there are three disadvantages for query processing based on index structures compared to the sequential scan:

- data is read in too small portions

- index structures have a substantially lower storage utilization
- processing of the directory causes overhead

In this chapter, we will present the DABS-tree (Dynamic Adaptation of the Block Size) which tackles all three problems. We propose a new index structure claiming to outperform the sequential scan in virtually every case. In dimensions where index-based techniques are superior to the sequential scan, the efficiency of these techniques is retained unchanged. In an area of moderate dimensionality, both approaches, conventional indexes as well as the scan, are outperformed.

The first problem is solved by a suitable page-size optimization. As we face the problem that the actual optimum of the logical block size is dependent on the number of objects currently stored in the database and on the data distribution (which may also change over time), the block size has to be adapted dynamically. After a page has been affected by a certain number of inserts or deletions, the page is checked whether the number of points currently stored in the page is close enough to the optimum. Otherwise, the page is split or a suitable partner is sought for balancing or merging.

This means that pages with different logical block size are at the same time stored in the index. Although a constant block size facilitates management, no principal problem arises when sacrificing this facilitation. To solve the second problem, storage utilization, we propose to allow continuously growing block sizes, i.e. we also give up the requirement that the logical block size is a multiple of some physical block size or a power of two or the demand that the block size is only changed by doubling or division by two. Instead, every page has exactly the size which is needed to store its current entries. When an entry is inserted to a page, the block size increases, and the page must usually be stored to a new position in the index file. To avoid fragmentation of the file, we propose garbage collection.

The third problem, directory overhead, cannot be completely avoided by our technique since we do not want to cancel the directory. The directory overhead, however, is weakened, because we simplify the directory. Instead of a hierarchical directory, we only maintain a linear single-level directory which is sequentially scanned. The block size optimization also helps to reduce the directory overhead, because this overhead is taken into account by the optimization.

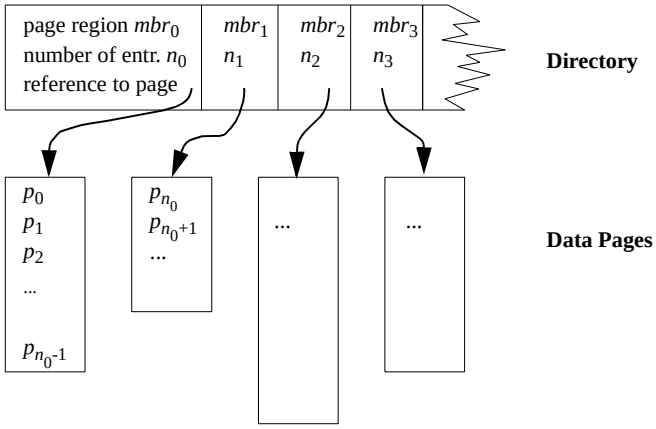


Figure 56: Structure of the DABS-Tree.

4.3 Structure of the DABS-Tree

The structure of the DABS-tree is depicted in figure 56. Each directory entry contains the following information: The page region in form of a minimum bounding rectangle, the reference (i.e. the background storage address) to the page and additionally the number of entries currently stored in the page. The number of entries is also used to determine the corresponding block size of a data page before loading.

The directory consists simply of a linear array of directory entries. We intentionally cancel the hierarchically organized directory, because the efficiency of query processing is not increased by hierarchies but rather decreased. We confirm this effect by the following consideration:

In our experiment presented in section 4.1 (cf. figure 55, too), we determined an optimum block size of 64 KBytes. For 10,000,000 data points in a 20-dimensional space, we need 20,000 data pages to store the points. Using a hierarchical directory, we need 78 index pages at the first directory level and the root-page. Even if we assume no overlap among the directory pages, query processing requires an average of 44 directory page accesses. The cost for these accesses are 1.14 seconds of I/O time. A sequential scan of

a linear directory, however, requires 0.71 seconds. Both kinds of directory cost are negligible compared to 49 seconds of cost for accessing the data pages. Even though, the sequential scan of a linear directory causes fewer effort than a hierarchical directory. This observation even holds for fairly low dimensions.

The data pages contain only data-specific information. Besides the point data and eventually some additional application-specific information, no management information is required. The data pages are stored in random order in the index file. Conventional index structures usually do not utilize the space in the data pages to 100% in order to leave empty space for future insert operations. In contrast, the DABS-tree stores the data pages generally without any empty position inside a data page and without any gap between different pages. Whenever a new entry is inserted to a data page, the page is stored at a new position. The empty space in the file where the data page formerly used to be is passed to a free memory management. A garbage collection strategy is applied to build larger blocks of free memory, and, thus to avoid fragmentation (cf. section 4.5). Temporarily, the free blocks decrease the storage utilization of the index structure below 100%. The free blocks, however, are never subject to a reading operation during insert processing. Therefore, the performance of query processing cannot be negatively affected.

In order to guarantee overlap-free page regions, we hold additionally to the linear directory a kd-tree [Ben 75, Ben 79]. As we explained in section 2.4.4, a kd-tree partitions the data space in a disjoint and overlap-free way. The page regions of the DABS-tree are always located inside a single kd-tree region. The kd-tree facilitates insert processing, because it offers unambiguously a data page for the insert operation. In contrast, the heuristics for choosing a suitable page in the X-tree cannot guarantee that no overlap occurs. The kd-tree is also used for the merging operation which may be necessary due to delete operations, or because the optimal page size has increased on the basis of a changed data distribution. The kd-tree is not used for search.

4.4 Search in the DABS-Tree

Point queries and *range queries* are handled in a straightforward way. First, the directory is sequentially scanned. All data pages qualifying for the query (i.e. containing the query point or intersecting with the query range, respectively) are determined, loaded and processed.

```
Point DABS_nearest_neighbor_query (Point q) {
    typedef struct {float distance, int pageno, int num_objects} AplEntry ;
    AplEntry apl [number_of_pages]
    int i, j;
    Point cpc ;
    float pruning_dist = +infinity ;

    // First Phase
    DIRECTORY dir = read_directory () ;
    for (i = 0 ; i < number_of_pages ; i ++) {
        apl [i] . distance = mindist (q, dir [i] . mbr) ;
        apl [i] . pageno = dir [i] . pageno ;
        apl [i] . num_objects = dir [i] . num_objects ;
    }
    qsort (apl, number_of_pages, sizeof (AplEntry), cmp_float) ;

    // Second Phase
    for (i = 0 ; i < number_of_pages && apl [i] . distance < pruning_dist ; i ++) {
        Page p = LoadData (apl [i] . pageno, apl [i] .num_objects) ;
        for (j = 0 ; j < apl [i] . num_objects ; j ++)
            if (dist (q, p . object [j] . point) < pruning_dist) {
                cpc = p . object [j] . point ;
                pruning_dist = dist (q, p . object [j] . point) ;
            }
    }
    return cpc ;
}
```

Figure 57: Algorithm for Exact Match Queries.

Nearest neighbor queries and *k-nearest neighbor queries* are processed by a variant of the HS algorithm (cf. section 2.3.4, [HS 95]). As the directory is flat, the algorithm can even be simplified, because the *active page list* (*APL*) is static in absence of a hierarchy. For hierarchically organized directories, query processing requires permanent insert operations to the APL, because in each processing step the pivot page is replaced by its child pages. Therefore, the APL must be re-sorted after processing a page.

In our case, the nearest neighbor algorithm works in two phases: The first phase scans the directory sequentially. During the scan, the distance between the query point and each page region is determined and stored in an array. Finally the distances in the array are sorted by the Quicksort algorithm, for instance [Hoa 62, Sed 78]. In the second

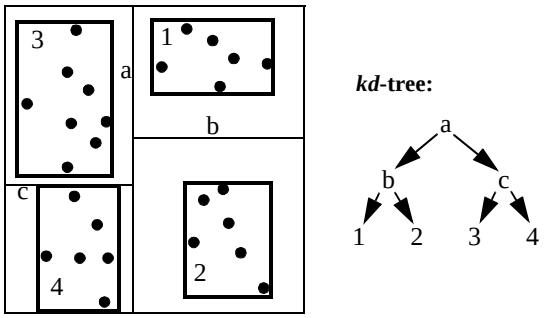


Figure 58: The Additional *kd*-tree.

phase, the data pages are loaded and processed in the order of increasing distances. The closest point candidate determines the pruning distance. Query processing stops when the current page region is farther away from the query point than the closest point candidate. Figure 57 depicts the algorithm for nearest neighbor queries. The *k-nearest neighbor* algorithm works analogously with the only difference that a *closest point candidate list* consisting of *k* entries is maintained and that the last entry in this list determines the pruning distance.

4.5 Handling Insert Operations

4.5.1 Searching the Data Page

To handle an insert operation, we search in the kd-tree, which is held in addition to the linear directory, for a suitable data page. The kd-tree has the advantage to partition the data space in a complete and disjoint fashion which makes the choice of the corresponding page unambiguous. Eventually, the MBR in the linear directory which is always located inside the corresponding kd-tree region (cf. figure 58) must be slightly enlarged.

The page is loaded to the main memory, and the point is inserted. Usually, the page cannot be stored at its old position since we enforce a 100% storage utilization of pages. Therefore, it is appended to the end of the index file. The empty block at the former position of the page is passed to a free storage manager which performs garbage collec-

tions if the overall storage utilization of the index file decreases below a certain threshold value (e.g. 90%).

Note that in contrast to conventional index structures, the overall storage utilization can never decrease the efficiency of query processing, because empty parts of the index file are not subject to reading operations. By a low storage utilization, we only waste storage memory, but not processing time.

4.5.2 Free Storage Management

The free storage manager currently observes the storage utilization of the index file. When the storage utilization reaches some threshold value $su_{\min}$, the next new page is not appended to the end of the index file. Instead, a local garbage collection is raised which performs local restructuring of the file to collect empty pages as follows:

Let the size of the next page to be stored be *s*. The storage manager searches for the shortest interval of subsequent pages in the index file covering *s* Bytes of empty space. With a suitable data structure to organize the empty space, this search can be performed in O (log *n*) time for the average case. Once the shortest interval with *s* Bytes of empty space is found, we load all pages in this interval to the main memory and restore them densely, thus creating a contiguous empty space of at least *s* Bytes. We store the new page to this space.

Now we will claim an important property of the restructuring action: Locality. We show that the size of the interval in the file which is to be restructured is bounded by the size *s* of the new page multiplied with some factor depending on the storage utilization $su_{\min}$.

Lemma 6: Locality of Restructuring

In an index file with a storage utilization $su \leq su_{\min}$, there exists an interval with the length

$$l = \frac{s}{1 - su_{\min}}$$

containing at least *s* Bytes of free storage.

Proof (Lemma 6)

Assume that all intervals of the length *l* have less than *s* Bytes of free storage. Then, the number *e* of free Bytes in the file with length *f* is bounded by:

$$e < \frac{f}{l} \cdot s$$

By the definition of the storage utilization, we get the following inequation

$$su = 1 - \frac{e}{f} > 1 - \frac{s}{l} = su_{\min}$$

which contradicts the initial condition $su \leq su_{\min}$.

❑

If we choose, for instance, a storage utilization of $su_{\min}$ = 50%, Lemma 6 tells us that restructuring is bounded to an interval twice as large as the size s of the page we want to store. For a storage utilization of $su_{\min}$ = 90%, the interval is at most ten times as large as the new page.

As there are no specific overflow conditions in our index structure, the pages are periodically checked by using a cost estimation whether they must be split. For the details, cf. section 4.7.

4.6 Handling Delete Operations

Deleting in the DABS-tree is straightforward. The point is deleted from the corresponding page and a small block is passed to the free storage manager. If the storage utilization falls below the threshold $su_{\min}$, a local restructuring action is raised for the last data page in the file.

Since there is no clear underflow condition in the DABS-tree, the pages are periodically tested by using a cost model whether they are to merge.

4.7 Dynamic Adaptation of the Block Size

In this section, we will first show the dynamic adaptation from an algorithmic point of view. Then, we will show how the cost model developed in chapter 3 is modified and applied to take split and merging decisions, respectively.

4.7.1 Split and Merge Management

Basically, it is possible to evaluate the cost model after every insert or delete operation and to determine whether a page must be split or merged with some neighbor. This is, however, not very economic, because the optimum is generally broad. Therefore, we have to check rather seldom if the current page size still is close to the optimum.

We choose the following strategy: For each page, we have an update counter variable which is increased in each insert or delete operation the page is subject to. We perform our model evaluations when the value of the update counter reaches some user defined threshold which may be defined as a fixed number (e.g. 20 operations) or as a ratio of the current page capacity (e.g. 25% of the points in the page).

Note that it is theoretically possible (although not very likely) that pages must be merged after performing insert operations or that pages must be split after performing delete operations. This is not intuitive, as conventional index structures with a fixed block size know to split only after inserts and to merge after deletions. In our dynamic optimization, however, any of these operations can change the distribution of the data points and thus change the page size optimum into each direction.

Whenever the threshold of update operations is reached, a cost estimate for the current page with respect to query processing is determined. Then, some split algorithm is run tentatively. The page regions of the created pages are determined, and the query processing cost for the new pages is estimated. If the performance has decreased, the split is undone, and merging is tested in the same way.

A merging operation can only be performed if a suitable partner is available. In order to maintain overlap-free page regions, only two leaf pages with a common parent node in the kd-tree are eligible for merging. If the current page does not have such a counterpart, merging is not considered. Otherwise, the cost estimates for the two single pages and for the resulting page are determined and compared. If the performance estimate improves, the merge is performed.

Finally, the relevant update counters are reset to 0.

4.7.2 Model Based Local Cost Estimation

For our local cost optimization, we must estimate how cost of query processing changes when performing some split or merge operation. Generally, we assume as reference query the nearest neighbor query with the maximum metric, because this assumption causes the lowest effort in the model computation. Practically, the difference in the page

size optimum is low when changing the reference query to the Euclidean metric or to some *k*-nearest neighbor query.

In both cases, when taking a split or a merge decision, we compare the cost caused by one page with the cost caused by two pages with the half capacity. At the one hand, this action changes the accessing cost, because the transfer cost decreases with decreasing capacity. The access probability is also decreased by splitting. At the other hand, it is unpredictable whether the sum of the costs caused by the two smaller pages is really lower than the cost of the larger page.

Therefore, it is reasonable, to draw the following balance for the split decision:

$$\Delta_T = (t_{\text{Seek}} + C_1 \cdot t_{\text{Point}}) \cdot X_1 + (t_{\text{Seek}} + C_2 \cdot t_{\text{Point}}) \cdot X_2 - (t_{\text{Seek}} + C_0 \cdot t_{\text{Point}}) \cdot X_0 ,$$

where C_0 and X_0 are the capacity and the access probability of the larger page, and C_1 and C_2 (X_1 and X_2) the capacities (access probabilities) of the two smaller pages. The time t_{Point} is the transfer time for a point, i.e. $t_{\text{Point}} = t_{\text{transfer}} \cdot \text{sizeof (Point)}$. If the cost balance Δ_T is positive, the larger page causes fewer cost than the two smaller pages. In this case, a split should be avoided and a merge should be performed.

It is possible to estimate the access probability according to our formulas for $X_{\text{nn,mm,ld,c}}$ and $X_{\text{nn,mm,hd,c}}$ presented in chapter 3. This approach, however, assumes no knowledge about the regions of the pages currently stored in the index. In our local cost optimization, the exact coordinates of the relevant page regions are known. Therefore, we can achieve higher accuracy if this information is considered. Additionally, it is possible to take into account the local exceptions in the data distribution.

First, we determine the local point density according to the volume and the capacity of the larger page:

$$\rho_F = \frac{C_0}{V(\text{MBR}_0)^{D_F/d}}$$

From the local point density, we can derive an estimation of the nearest neighbor distance:

$$r = \frac{1}{2} \cdot \sqrt[D_F]{\frac{1}{\rho_F}}$$

Here we apply here the simple model, because in this context, efficiency of evaluation is of higher importance than accuracy. Now, we are able to determine the Minkowski sum of the nearest neighbor query and the page region. If MBR_0 is given by a vector of lower

bounds (lb_0, ... $lb_{d\text{-}1}$) and upper bounds (ub_0, ... $ub_{d\text{-}1}$), the Minkowski sum is determined by:

$$V_{R \oplus C}(\mathrm{MBR}_0, 2r) = \prod_{0 \le i < d} (ub_i - lb_i + 2r)$$

This Minkowski sum can be explicitly clipped at the data space boundary (here for simplicity assumed to be the unit hypercube):

$$V_{(R \oplus C) \cap DS}(\mathrm{MBR}_0, 2r) = \prod_{0 \le i < d} (\min\{ub_i + r, 1\} - \max\{lb_i - r, 0\})$$

We assume that the query distribution follows the data distribution. Therefore, the access probability X_0 corresponds to the ratio of points in the Minkowski sum with respect to all points in the database:

$$X_0 = \frac{\rho_F}{N} \cdot V_{(R \oplus C) \cap DS}(\mathrm{MBR}_0, 2r)^{D_F/d}$$

Analogously, the access probabilities for the smaller pages X_1 and X_2 are determined by their page regions MBR_1 and MBR_2. The access probabilities are used in the cost balance for taking split or merge decisions.

4.7.3 Monotonicity Properties of Splitting and Merging

The most important precondition for the correctness of a local optimization is the monotonicity of the first derivative of the cost function with respect to the page capacity. If the first derivative is not monotonically increasing, the cost function may have various local optima where the optimization easily could get caught in.

As depicted in figure 55, the cost function indeed forms a single local optimum which is also the global optimum. Cost are very high for block sizes which are either too small or too large. Minimum cost arise in a relatively broad area between these extremes.

Under several simplifying assumptions, it is also possible to prove that the derivative of the cost function is monotonically increasing. From this monotonicity, we can conclude that there is at most one local minimum. The assumptions required for this proof are uniformity and independence as well as neglecting boundary effects. For this simplified model

$$T(C) = \left(\sqrt[d]{\frac{1}{C}} + 1\right)^d \cdot \left(t_{\mathrm{Seek}} + \frac{C}{\mathrm{sizeof(point)}} \cdot t_{\mathrm{transfer}}\right),$$

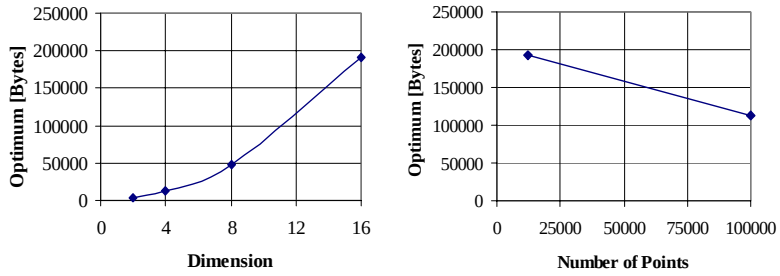


Figure 59: Optimal block size for Uniform Data.

it is possible to show that the second derivative of the cost function is positive:

$$\frac{\partial^2}{\partial C^2} T(C) \geq 0 .$$

The intermediate results in this proof, however, are very complex and thus not presented here.

4.8 Experimental Evaluation

To demonstrate the applicability and the practical relevance of our technique, we performed an experimental evaluation on both, synthetic and real data. The improvement potential was already shown in figure 55 where a clear optimum for page sizes was found at 64 KBytes outperforming the X-tree with a standard page of 4K by a factor of 2.7 and the sequential scan by a factor of 3.6.

The intention of our next experiment is to show that the optimum is not merely a hardware constant but to a large extent dependent on the data to be indexed. For this purpose, we constructed a DABS-tree on several data files containing uniformly and independently distributed points of varying dimension. The number of objects was fixed in this experiment to 12,000. We observed the block size which was generated by the local optimization. The results are depicted on the left side of figure 59. In the two-dimensional case, quite a usual block size of 3,000 Bytes was found to be optimal. In the high-dimensional case, however, the optimum block size reaches values up to 192 KBytes with even increasing tendency.

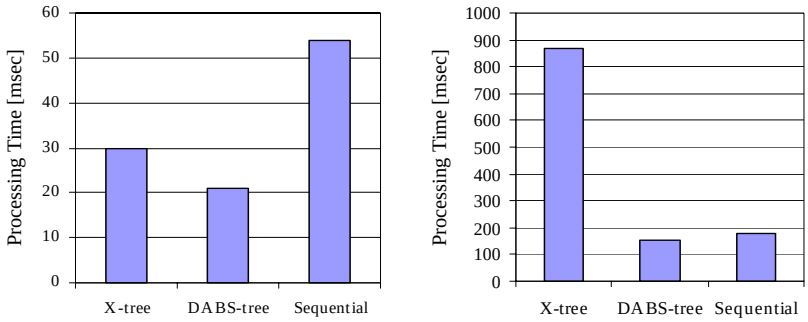


Figure 60: Performance for 4-Dimensional (left) and 16-Dimensional (right) Data.

In our next experiment, depicted on the right side of figure 59, we show the usefulness of dynamic optimization. We used the 16-dimensional index of the preceding experiments and increased the number of objects to 100,000. Hereby, the optimum page size decreased from 192 KBytes to 112 KBytes.

In our next experiment, depicted in figure 60, we compared the DABS-tree with the X-tree and the sequential scan. As expected, the performance in low-dimensional cases is similar to the X-tree; in high-dimensional cases it is similar to the sequential scan. In any case, both approaches are clearly outperformed. In the 4-dimensional example, the DABS-tree is 43% faster than the X-tree and 157% faster than the sequential scan. In the 16-dimensional example, the DABS-tree outperforms the sequential scan by 17% and the X-tree by 462%.

In case of a moderate dimensionality, and provided that the number of points stored in the databases is high, both techniques, the X-tree as well as the sequential scan, are clearly outperformed. This is demonstrated in the example of our 16-dimensional database with 100,000 points. Here, the improvement factor over the X-tree is 2.78. The improvement over the sequential scan is with 2.44 in the same order of magnitude.

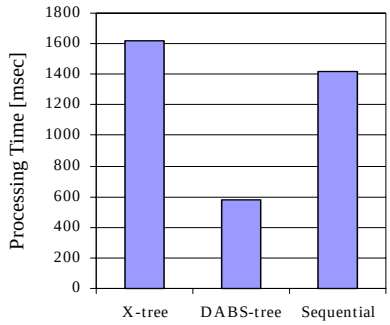


Figure 61: Sequential Scan and X-tree are Outperformed.

The intention of our last experiment is to confirm that our optimization technique is also applicable to real data and that high performance gains are reachable. For this purpose, we constructed a DABS-tree with 50,000 points from our CAD application (cf. section 1.1.1). We measured again the performance of nearest neighbor queries. As query points, we also used points from the same application which were not stored in the database. The data space dimension was 16 in this example. We outperformed the X-tree by a factor of 2.8 and the sequential scan by 6.6.

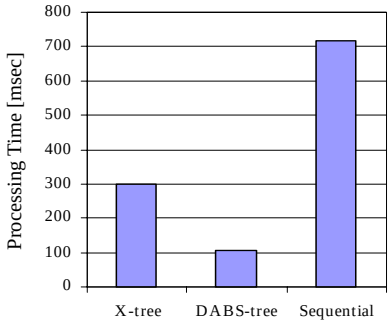


Figure 62: Query Processing Using CAD Data.

Chapter 5
Optimizing the Dimension Assignment

5.1 Introduction

One of the simplest techniques for multidimensional indexing is the inverted-list-approach. The basic idea of *inverted lists* is to use a one-dimensional index such as the B-tree [BM 72] or one of its variants for each attribute. In order to answer a given range query with *s* attributes specified, it is necessary to access *s* one-dimensional indexes and to perform a costly merge of the partial results obtained from the one-dimensional indexes. Inverted lists are available in most commercial database systems and thus are widely applied. For queries involving many attributes, however, the merging step is prohibitively expensive and is the major drawback of the inverted-lists approach.

It is well-known that multidimensional index structures are very efficient for databases with a small number of attributes and outperform inverted lists if the query involves multiple attributes [Kri 84]. In many real-life database applications, however, we have to handle databases with a large number of attributes. For databases with a larger number of attributes, the performance of traditional multidimensional index structures rapidly deteriorates. Therefore, specific index structures for high-dimensional data have been proposed. For high dimensions (larger than 12), however, even the performance of specialized high-dimensional index structures decreases.

In this chapter, we propose a new approach, called *tree striping*, for an efficient multi-attribute retrieval. The basic idea of tree striping is to divide the data space into disjoint subspaces of lower dimensionality such that the cross-product of the subspaces is the original data space. The subspaces are organized by using an arbitrary multidimensional index structure. Tree striping is a generalization of the inverted lists and multidimensional indexing approaches.

The rest of this chapter is organized as follows: Section 5.2 introduces the basic idea of tree striping including the algorithm necessary for query processing. Then in section 5.3, we provide a theoretical analysis of our technique and show that optimal query processing is obtained for tree striping. We also show that optimal tree striping outperforms the traditional inverted lists and multidimensional indexing methods. In section 5.4, we then discuss the more elaborate query processing algorithms which make use of the specific advantages of "striped" trees and therefore further improve the performance. Section 5.5 provides the details of our experimental evaluation which includes comparisons of tree striping to inverted lists and two multidimensional index structures, namely the R-tree and the X-tree. The results of our experimental analysis confirm the theoretical results and show substantial improvements over the multidimensional indexing and the inverted-lists approaches.

5.2 Tree Striping

Our new idea presented in this chapter is to use the benefits of both the inverted lists and high-dimensional indexing approaches in order to achieve an optimal multidimensional query processing. Our approach, called *tree-striping*, generalizes both previous approaches. A first experiment on uniformly distributed 16-dimensional data presented in figure 63 shows significant improvement factors of our method over the inverted lists and multidimensional indexing approaches. Our comprehensive experimental evaluation in section 5.5 will partly yield even more impressive improvement factors.

5.2.1 Basic Idea

The basic idea of tree-striping is to divide the data space into disjoint subspaces of lower dimensionality such that the cross-product of the subspaces is the original data space. (Note that a division of the data space into disjoint subspaces is different from a partitioning of the data space where the partitions have the same dimensionality as the origi-

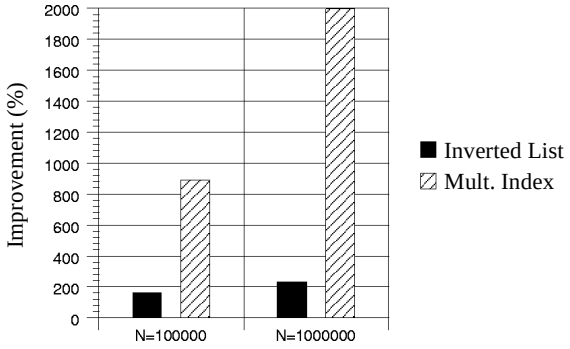


Figure 63: Improvement over Inverted Lists and Multidimensional Indexing

nal data space whereas subspaces have a lower dimensionality.) This means that each subspace contains a number of attributes (dimensions) and each object of the database occurs in all subspaces. For example, the three-dimensional data space (customer_no, discount, turnover) may be divided into the one-dimensional subspace (customer_no) and the two-dimensional subspace (discount, turnover). Obviously, the dimensionality of the subspaces is smaller than the dimensionality of the data space, and hence, we are able to index the subspaces more efficiently using any multidimensional index structure.

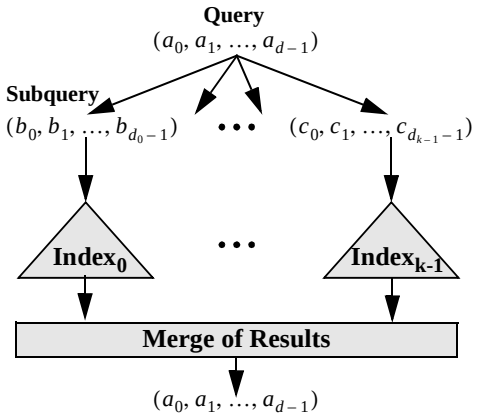


Figure 64: Tree Striping

To insert an object, we divide the object into subobjects according to the division of the data space. Then, we insert the subobjects in the multidimensional index structure managing the corresponding subspace. To process a query, we divide the query according to the division of the data space and issue the subqueries to the relevant multidimensional indexes. In a second step, we merge the results which have been produced by the indexes using an external sorting algorithm such as merge sort. The general idea and query processing strategy of tree striping is presented in figure 64.

Note that, in contrast to inverted lists, in general, the selectivity of subspace indexes is relatively high because each index manages information about more than one attribute. Therefore, the amount of partial results produced in the first step is rather small which means that the cost for the merging step are not significant. Our formal model which will be presented in section 5.3, confirms this fact.

It is clear that the number and dimensionality of the data space divisions are important parameters for the performance of our technique. The optimal division mainly depends on the dimension, the number of data items, and the data distribution. The parameters have to be chosen adequately to achieve an optimal performance. For a uniform data distribution, the parameters for an optimal division into subspaces can be obtained easily from the theoretical analysis (cf. section 5.3).

5.2.2 Definition of Tree Striping

In this section, we formally define the tree striping technique. In the following, we consider objects as vectors in a vector space and attributes as components of the vectors. Given is a data space of dimension d and extension $[0..1]^d$, N vectors v having the components $v_0 ... v_{d-1}$ and an arbitrary multidimensional index structure *MIS* supporting the relevant query types. First, we need a mapping which assigns the dimensions to the different subtrees.

Definition 10: Dimension Assignment

The dimension assignment *DA* is a mapping $R^d \rightarrow (R^{d_0}, \ldots, R^{d_{k-1}})$ of a d-dimensional vector v to a vector of k d_l-dimensional vectors w^l, $0 \leq l < k$, such that the following conditions hold:

$$1) \sum_{l=0}^{k-1} d_l = d$$

2) $\forall j \;\; 0 \le j < d, \;\; \exists l \;\; 0 \le l < k, \;\; \exists i \; 0 \le i < d_l : \;\; v_j = w_i^l$

3) $\forall l \;\; 0 \le l < k, \;\; \forall i \;\; 0 \le i < d_l, \;\; \exists j \; 0 \le j < d : \;\; w_i^l = v_j$

Note that w_i^l denotes the *i*-th component in the *l*-th index. To clarify the definition of dimension assignment, we provide a simple example: Given a 5-dimensional data space (*d*=5). We may define a dimension assignment DA_{odd_even} such that $k = 2$, $d_0 = 3$, and $d_1 = 2$, i.e. DA_{odd_even} divides the data space into two subspaces of dimensionality 3 and 2. Explicitly, DA_{odd_even} maps even dimensions to the first subspace and odd dimensions to the second subspace, more formally, $DA_{odd\_even}(v) = (w_0, w_1)$, $w_0 = (v_0, v_2, v_4)$, $w_1 = (v_1, v_3)$. Thus, a vector $v = (0, 4, 6, 5, 1)$ is mapped to $DA_{odd\_even}((0, 4, 6, 5, 1)) = ((0, 6, 1), (4, 5))$. Obviously, DA_{odd_even} meets the conditions specified in definition 10 because all dimensions of the data space have been mapped to a subspace and vice versa.

Using the definition of dimension assignment, we are now able to formally define tree-striping:

Definition 11: Tree Striping

Given a database *DB* of *N* *d*-dimensional vectors and a dimension assignment *DA*. Then, a tree-striping *TS* is defined as a vector of k d_l-dimensional indexes

$$MIS^l = \{w^l\}, \; 0 \le l < k,$$

with $w^l = DA^l(v), \; v \in DB$.

Tree striping as defined in definition 11 is a generalization of the previous approaches. For the special case of $k = d$, tree striping corresponds to inverted lists because the dimension assignment produces *d* one-dimensional data objects; and for the special case of $k = 1$, tree striping corresponds to the traditional multidimensional indexing approach because we have one *d*-dimensional index. The most important question is whether there exists a tree striping which provides better results than the extremes (the well-known inverted lists and multidimensional indexing approaches). In particular, we have to determine whether there exists a k ($1 < k < d$) such that tree striping outperforms the other approaches. In the next section, we introduce a theoretical model showing that an optimal k exists. Our experimental analysis presented in section 5.5 confirms the results of our theoretical model and shows performance improvements of up to a factor of 120 times over the inverted lists and up to 280% over the multidimensional indexing approach. A second open question is how the attributes (dimensions) are assigned to the

```
SetOfObject query(TreeStrip ts, QuerySpec qs)
{
  int i;
  SetOfSubObject sst[ts.num];
  SubQuerySpec sqs[ts.num];
  SetOfObject st;

  // for all indexes
  for (i = 0; i < ts.num; i++)
  {
    // query i-th index with sub-query
    sqs[i] = ts.opt_dim_assign(i, qs);
    sst[i] = ts.index[i].query(sqs[i]);
    // sort result by primary key
    sst[i].sort();
  }
  // now merge single results
  st = merge(sst, ts.num);
  return st;
}
```

Figure 65: A First Query Processing Algorithm

different trees such that the performance improvement is optimal. In section 5.4, we discuss the implications of different dimension assignments and also introduce optimized algorithms for query processing using striped trees.

Note that tree striping as defined so far is independent of the multidimensional index structure used. Any multidimensional index structure such as the R-tree [Gut 84] and its variants (R^+-tree [SRF 87], R^*-tree [BKSS 90], P-tree [Jag 90b]), Buddy-tree [SK 90], linear quadtrees [Gar 82], z-ordering [Ore 90] or other space-filling curves [Jag 90], and grid-file based methods [NHS 84, Fre 87] may be used for this purpose.

Before we describe our theoretical model, we first provide a simple algorithm for processing queries using striped trees. As the single indexes do not have all information about an object, but only about some attributes of the object, in general, we have to query all indexes in order to process a query. Therefore, we divide the query specification *qs* into sub-query specifications *sqs*[*l*] according to the dimension assignment. Then, we query each single index with the sub-query specification *sqs*[*l*] and record the results. In a final step, we have to merge the results by sorting the single results according to the primary key of the objects or any object identifier. Figure 65 shows a first version of a query processing algorithm. An optimized version for querying striped trees is provided in section 5.4.

5.3 Analytical Model

As already mentioned, most of the multidimensional indexing approaches efficiently solve the multi-attribute retrieval problem on low-dimensional data. From our experience in real-life database projects, we have learned that even for relational database systems handling relatively high numbers of attributes (more than 10) is necessary, for which the performance of traditional index structures deteriorates. To process arbitrary queries (e.g., point, range, and partial match queries) efficiently on those databases, we have to equally index all the attributes which means that we have to deal with a high-dimensional data space.

Unfortunately, some mathematical problems arise in high-dimensional spaces which are usually summarized by the term 'curse of dimensionality.' A basic effect in high-dimensional space is the exponential growth of the volume: Let us assume a database of 1,000,000 uniformly distributed objects consisting of 20 numerical attributes in the range [0...1]. Let us further assume that we are interested in a query which provides 10 result objects located around the midpoint of the data space (0.5, 0.5, 0.5, ..., 0.5). Which range do we have to query in order to obtain 10 result objects? Obviously, we have to assure that the volume of our query range is equal to

$$\frac{10}{1{,}000{,}000} = 10^{-5},$$

as the volume of the data space is equal to 1. This leads to a query range of

$$\sqrt[20]{10^{-5}} \approx 0.56$$

in each attribute. So we have to query the range (0.22-0.78, 0.22-0.78, ..., 0.22-0.78). That means a query with a selectivity of 10^{-5} leads to a query range of 0.56 in each attribute in a 20-dimensional data space.

Considering these effects, we are able to provide a concise cost model of processing range queries in a high-dimensional data space using the tree striping technique. For the following, we assume a uniformly distributed set of N vectors in a d-dimensional space of extension $[0..1]^d$. Note that even if we assume a uniform distribution of the data, our model can be applied to real data as well (cf. section 5.5). We will use the cost model to determine the optimal number of trees and accordingly the dimensions of the trees for a given data set, i.e. the optimal dimension assignment.

Our cost model is divided into two parts: First, the cost arising from querying the striped trees, and second, the cost for merging the results of the striped trees into one final result.

Both cost functions are highly influenced by the dimensions of the striped trees. The lookup cost in the index is growing superlinearly with growing tree dimension. The merging cost, however, is growing superlinearly with the size of the result which is, in turn, falling with the dimension of the trees. This fact implies the assumption that the total cost could form a minimum where both costs are moderate. This minimum should be located anywhere between the *d*-dimensional index and the inverted-lists approaches.

We assume that the multidimensional index structure aggregates a fixed number of C_{eff} vectors into a data page such that the bounding box containing the vectors forms a square-shaped hyperrectangle with the (hyper-) volume

$$V_{BB} = \frac{C_{eff}(d)}{N}.$$

Thus, C_{eff} denotes the actual fan-out of the index. From that, the edge length σ of a typical bounding box is

$$\sigma = \sqrt[d]{V_{BB}} = \sqrt[d]{\frac{C_{eff}(d)}{N}}.$$

Analogously, we compute the edge length q of V_q as $q = \sqrt[d]{V_q}$. The expected number of page accesses $A_{\text{index}}(d,N,q)$ is determined according to our cost model in chapter 3:

$$\begin{aligned} A_{\text{index}}(d, N, q) &= \frac{C_{eff}(d)}{N} \cdot \left(\sqrt[d]{\frac{C_{eff}(d)}{N}} + q \right)^d \\ &= \left(1 + q \cdot \sqrt[d]{\frac{N}{C_{eff}(d)}} \right)^d \end{aligned}$$

The number C_{eff} of data vectors in a data page also depends on the dimension d of the vectors. Assuming that each coordinate value is stored as a 32-bit floating point value and that there is an additional unique object identifier which also requires 32 bit, we determine C_{eff} as:

$$C_{eff} = \frac{\text{pagesize} \cdot \text{storageutilization}}{4 \cdot (d + 1)}$$

The cost for combining the results of the multidimensional index accesses mostly depend on the selectivities of the indexes. If |FRS| is the size of the final result set of query

Q then $|IRS_i|$ is the intermediate result set produced by the *i*-th index having dimension d_i. Thus:

$$\frac{|IRS_i|}{N} = \left(\frac{|FRS|}{N}\right)^{\frac{d_i}{d}} = q^{d_i}$$

Note that we have to sort each intermediate result set according to the object identifiers in order to be able to merge them into the final result set. We have to apply an external sorting algorithm since, for larger q or minor d_i, the result set will exceed the available main memory. According to Ullman [Ull89], the cost for performing multi-way merge-sort on a relation of B blocks is $2B \cdot \log_M(B)$ where M is the number of cache pages available to the sorting process. We can store the object identifiers in a densely packed fashion such that $|IRS_i|$ object identifiers require

$$\frac{4 \cdot |IRS_i|}{\text{page-size}}$$

pages. From that, the cost for sorting the result set of a single index are:

$$A_{sort}(d_i, N, q) = \frac{8 \cdot N \cdot q^{d_i}}{\text{page-size}} \cdot \log_M\left(\frac{(4 \cdot N \cdot q^{d_i})}{\text{page-size}}\right)$$

To determine the total cost, A_{sort} and A_{index} have to be summed up for all striped trees. For merging the result sets, each of them has to be scanned once more. Total cost is:

$$A(d_i, N, q) = \sum_{i=1}^{k} \left[\left(1 + q \cdot \sqrt[d_i]{\frac{N}{C_{eff}(d)}}\right)^{d_i} + \frac{4 \cdot N \cdot q^{d_i}}{\text{page-size}} \cdot \left(1 + 2 \cdot \log_M\left(\frac{(4 \cdot N \cdot q^{d_i})}{\text{page-size}}\right)\right)\right]$$

In the following, we assume that the d dimensions of our data space are striped into k divisions:

$$d_0 = d_1 = ... = d_{k-1} = \frac{d}{k}$$

For d_0, ..., d_{k-1}, only whole numbers are meaningful. This effect is handled later, but is of minor importance for our cost model. In this case, our cost function can be simplified to:

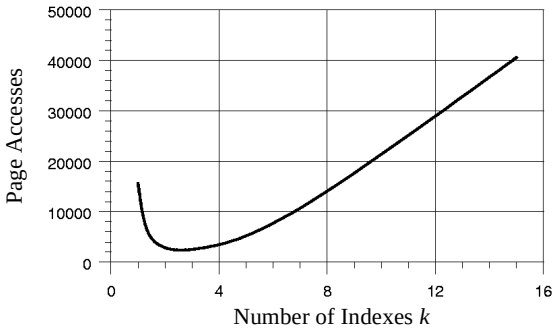


Figure 66: Total Cost for Query Processing

$$A(d, k, N, q) = k \cdot \left[\left(1 + q \cdot \left(\frac{N}{C_{eff}(d/k)}\right)^{k/d}\right)^{d/k} + \right.$$
$$\left. + \frac{4 \cdot N \cdot q^{d/k}}{\text{page-size}} \cdot \left(1 + 2 \cdot \log_M\left(\frac{4 \cdot N \cdot q^{d/k}}{\text{page-size}}\right)\right)\right].$$

Figure 66 shows the total cost over k in a typical setting with a database of 1,000,000 uniformly distributed objects in a 15-dimensional data space. The selectivity of the query is 0.01%. There is a clear minimum between k=2 and k=3.

Thus, we are able to determine an optimal k by solving the following equation:

$$\frac{\partial}{\partial k} A(d, k, N, q) = 0 \qquad (*)$$

The analytic evaluation of this equation yields a rather large formula which is omitted due to space limitations. A function in the C language automatically generated by some math program (MATHEMATICA, MAPLE) determining the derivative can be used to calculate the optimum.

Unfortunately, the cost model presented so far is accurate only in the low-dimensional case. This is caused by the fact that in high-dimensional data spaces the data pages cannot be split in each dimension. If we split a 20-dimensional data space once per dimension, we obtain 2^{20}=1,000,000 data pages. Obviously, the number of data objects would have to grow exponentially with the dimension in order to allow one split per dimension. Therefore, we provide a special high-dimensional adaptation of our cost

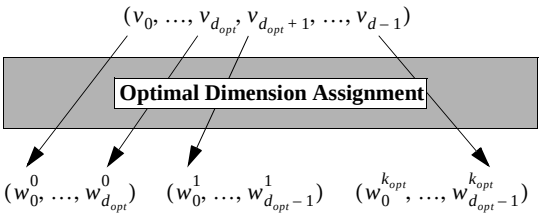


Figure 67: Optimal Dimension Assignment

model. Our extension assumes that data pages are split only in the first *d'* dimensions where *d'* is the logarithm of the number of data pages to the basis of two:

$$d' = \log_2\left(\frac{N}{C_{eff}(d_i)}\right).$$

The data pages have the average extension 1/2 in *d'* dimensions and extension 1 in all remaining dimensions (*d*-*d'*). When determining the Minkowski sum, we additionally have to consider that only a part of the volume is located inside the data space because in the dimensions which have not been split, the extension of the Minkowski sum is still 1 rather than (1+*q*):

$$HiDiMink(V_{BB}, V_q) = \left(\frac{1}{2} + \frac{q}{2}\right)^{d'} \cdot 1^{d-d'}.$$

Thus, the expected number of data pages accessed in the high-dimensional case is:

$$A_{\text{index, HiDi}}(d, N, q) = \frac{N}{C_{eff}(d_i)} \cdot \left(\frac{1}{2} + \frac{q}{2}\right)^{\log_2\left(\frac{N}{C_{eff}(d)}\right)}.$$

Adding the sort cost we obtain the following total cost for high-dimensional data spaces:

$$A_{\text{HiDi}}(d, k, N, q) = k \cdot \left[\frac{N}{C_{eff}(d_i)} \cdot \left(\frac{1}{2} + \frac{q}{2}\right)^{\log_2\left(\frac{N}{C_{eff}(d_i)}\right)} + \right.$$
$$\left. + \frac{4 \cdot N \cdot q^{d/k}}{\text{page-size}} \cdot \left(1 + 2 \cdot \log_M\left(\frac{4 \cdot N \cdot q^{d/k}}{\text{page-size}}\right)\right) \right].$$

5.4 Query processing

For optimal response times, we have to make two decision: We first have to choose an adequate dimension assignment and second, we have to choose the right strategy for processing queries.

As a result of the theoretical analysis presented in section 5.3, there exists an optimal number k of striped trees which can be determined according to our cost model (cf. equation (*)). Since k is a real number, however, we cannot directly use k as a parameter for our query processor. Instead, we use the floor of k

$$k_{opt} = \lfloor k \rfloor$$

and then determine the optimal dimensionality of our trees given by

$$d_{opt} = \lfloor d/k \rfloor .$$

Since in general, $(k_{opt} \cdot d_{opt})$ is smaller than d, we have to distribute the remaining

$$d_{rem} = d - (k_{opt} \cdot d_{opt})$$

attributes to our trees. Thus, we obtain d_{rem} trees with dimensionality $(d_{opt} + 1)$ and $(k_{opt} - d_{rem})$ trees with dimensionality d_{opt}. In the following, we have to distinguish between two cases: The first case is that we have additional information about the selectivity of the attributes which usually occurs for relational databases. The second case is that we have no additional information which usually occurs in indexing multimedia data using feature vectors. Let us first consider the more general case that we do not have any additional information and therefore assume that all attributes have the same selectivity. In this case, the optimal dimensionality d_{opt} of our trees may be used to define the following Optimal Dimension Assignment.

Definition 12: Optimal Dimension Assignment

The dimension assignment DA_{opt} is a dimension assignment according to definition 10 such that:

$$DA^l(v)_i = w_i^l = \begin{cases} v_{l(d_{opt}+1)+i} & \text{if } l < d_{rem} \\ v_{d_{rem}(d_{opt}+1)+(l-d_{rem})d_{opt}+i} & \text{otherwise} \end{cases},$$

where $0 \le l < k_{opt}$, $d_l = \begin{cases} d_{opt}+1 & \text{if } l < d_{rem} \\ d_{opt} & \text{otherwise} \end{cases}$, and $0 \le i < d_l$.

```
void insert(TreeStrip ts, object t)
{ int l;
  SubObject st[ts.num];

  // for all indexes
  for (l = 0; l < ts.num; l++)
  { // determine sub-objects
    st[l] = ts.opt_dim_assign(l, t);
    // insert sub-objects into l-th index
    ts.index[l].insert(st[l]);
  }
}
```

Figure 68: Insertion Algorithm

Intuitively, the optimal dimension assignment assigns the *i*-th component of the original vector *v* to a component of one of the vectors w^l such that the first vector w^0 receives the first d_0 components $(v_0 \ldots v_{d_0-1})$, the second vector w^1 accommodates the components $(v_{d_0} \ldots v_{d_0+d_1-1})$ and so on.

Using the optimal dimension assignment according to definition 12, now we are able to present the insert algorithm of our tree striping technique as depicted in figure 68. In order to insert an object *t*, we simply divide *t* into a set of k_{opt} sub-objects *st*[*l*] (using the optimal dimension assignment) and insert them into the according striped tree *ts.index*[*l*] $(0 \le l < k_{opt})$.

A more complex algorithm is required for processing queries on striped trees. A rather simple query processing algorithm has already been presented in section 5.2. The algorithm depicted in figure 65, however, has a major drawback: Let us assume that we have to process a partial range query *PRQ* which specifies the attributes *a*, *b* and *c*:

$$PRQ = \{*,*,[a_l, a_u], *,*,[b_l, b_u], *,*,[c_l, c_u], *,*\}.$$

Let us further assume that all these three attributes are located in the first of the striped trees. Obviously, it does not make sense to query any tree other than the first tree because all other trees do not have any selectivity. The algorithm presented in figure 65, however, executes queries on all trees ignoring the expected selectivity of the trees. In order to process queries efficiently, we have to take the selectivity of a tree into account and query a tree only if the expected gain in selectivity is worth the cost of querying the tree.

Another potential improvement of the query processing algorithm can be exemplified by the following situation: Assume that the three specified attributes *a*, *b* and *c* in the

```
SetOfObject query(TreeStrip ts, QuerySpec qs)
{
   int i, cost_index, cost_linear;
   SetOfSubObject sst[ts.num];
   SubQuerySpec sqs[ts.num];
   SetOfObject st; // set of candidates

   // sort indexes according to selectivity
   ts.sort_index(qs);

   // determine sub-queries
   for (i = 0; i < ts.num; i++)
      sqs[i] = ts.opt_dim_assign(i, qs);

   i = 0;
   // estimate cost
   cost_index = cost_modell(sqs[0]);
   cost_linear = cost_linear_scan(sqs[0]);
   while (i < ts.num &&
         cost_index < cost_linear)
   {  // query index
      sst[i] = ts.index[i].query(sqs[i]);
      // sorted merge of result
      sst[i].sort();
      merge(st, sst, ts.num);

      // estimate cost
      if (i < ts.num)
      {  cost_index = cost_modell(st, sqs[i+1]);
         cost_linear = cost_linear(st);
      }
   }

   if (i < ts.num)
   {  // load attributes
      database.load(st);
      remove_false_hits(st, qs);
   }

   return st;
}
```

Figure 69: Query Processing Using Tree Striping

above example are spread over two striped trees (T_0 managing attributes *a* and *b*, and T_1 managing attribute *c*). After querying tree T_0, we will typically receive a set of answers (candidates) which may contain some false hits. This assumption holds because the selectivity of T_0 is much higher than the selectivity of T_1. If we furthermore assume to have meaningful queries, i.e. queries having a good selectivity on all attributes, in general the set of candidates will be small. In this case, the cost for loading the candidate objects

from the secondary storage and checking if the objects fulfill the query specification may be lower than the cost of querying additional trees.

Let us now consider the second case where we do have some additional information about the selectivity of the attributes. A different selectivity of the attributes may be induced by the attributes of different data types (e.g., a Boolean attribute usually has a selectivity of 50%) and by different data distributions. We can use this information to adapt the optimal dimension assignment. If we are able to query the tree containing the attributes with the highest selectivity first, the resulting set of candidates will be rather small and will contain only a few false hits. Therefore, query processing can be finished without querying the other trees. This means that if we have information about the selectivity of attributes, we should sort the attributes according to their selectivity before applying the dimension assignment. Note that this operation does not only involve query processing but also the dimension assignment since we have to ensure that the attributes with the best selectivity are assigned to the first trees. In some cases, a non-uniform division may lead to better results. For example, let us assume that we have objects with 9 attributes (*a*, *b*, ... *i*) that k_{opt} is equal to 3, and that the attributes *a* to *d* have a high selectivity whereas the selectivity of attributes *e* to *i* is rather low. Then, it is beneficial to divide the objects into sub-objects (*a*, *b*, *c*, *d*), (*e*, *f*), and (*g*, *h*, *i*) which would be a sub-optimal division assuming no *a priori* knowledge about the selectivity of attributes.

Considering all these effects, we are now able to provide a more sophisticated algorithm for query processing on striped trees. The algorithm (cf. figure 69) first determines whether a linear search of the database is expected to be cheaper than a search using trees which may be the case for very large queries. The algorithm then sorts the striped trees according to their selectivity, i.e. the tree which probably provides the smallest set of candidates is queried first. If the querying of the first tree leads to a small set of candidates, we determine whether loading these candidates from the secondary storage is cheaper than querying the second tree. If this is the case, we load the attributes and output all candidates fulfilling the query specification. Otherwise, we query the second tree. This process iterates until all trees have been queried or the candidates are loaded and processed.

As the implementation of multidimensional index structures is complex, the assignment of different data types such as strings and floating numbers into one tree is not practicable. The division of a object may therefore be induced not only by the expected performance improvement but also by other considerations. Obviously, this can lead to

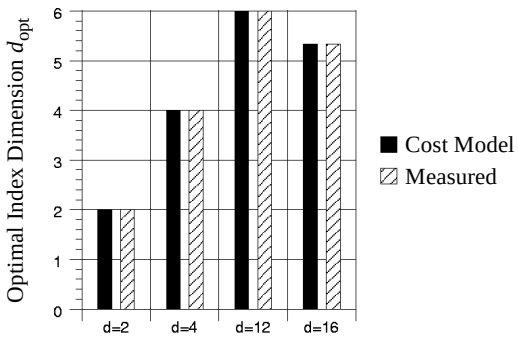


Figure 70: Comparison of Measured Optimum and Model Prediction

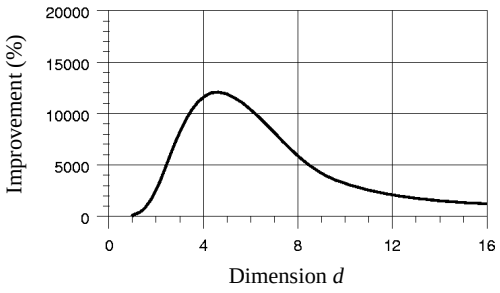
sub-optimal dimension assignments. Our practical experience, however, shows that a slightly sub-optimal dimension assignment performs nearly as well as the optimal dimension assignment.

5.5 Experimental Analysis

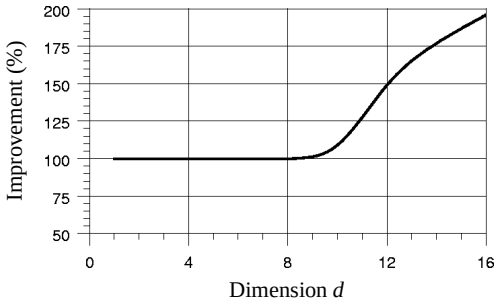
To show the practical relevance of our method, we performed an extensive experimental evaluation of tree striping and compared it to the inverted lists and the multidimensional indexing approach. All experimental results have been computed on an HP9000/780 workstation with several GBytes of secondary storage. For the experiments, we used an object-oriented implementation (C++) of the R*-tree [BKSS 90] and the X-tree [BKK 96].

The test data used for the experiments are real data consisting of text data describing substrings of a large database of texts, and synthetic data consisting of uniformly distributed points in high-dimensional space. The block size used for our experiments is 4 KByte, and all query processing techniques were allowed to use the same amount of cache. For a realistic evaluation, we used very large amounts of data (up to 80 MBytes) in our experiments. The total amount of disk space occupied by the created indexes (inverted lists, multidimensional indexes and tree-striped indexes) is about 2 GByte and the CPU-time for inserting the data adds up to about one week.

In a first experiment, we confirmed our theoretical result (cf. section 5.3) that the tree striping technique as a generalization of the lists and multidimensional indexing approaches outperforms both other techniques. For the experiment, we used 1,000,000 uniformly distributed data objects of varying dimensionality (d = 2..16). We built the according indexes (R*-tree) and queried the indexes with a selectivity of 10^{-5} which corresponds to an expected result of about 10 hits. In order to avoid statistical effects, we used the average cost of 100 uniformly distributed query windows. The observed variance was rather small. We compared different tree stripings (varying the value of k) and determined the optimal dimension assignment (optimal value of k). The tested dimension assignments for the 16-dimensional data set are (16), (8, 8), (6, 5, 5), (4, 4, 4, 4), (2, 2, 2, 2, 2, 2, 2, 2), and (1, 1, ..., 1, 1). The data sets of other dimensionality have been



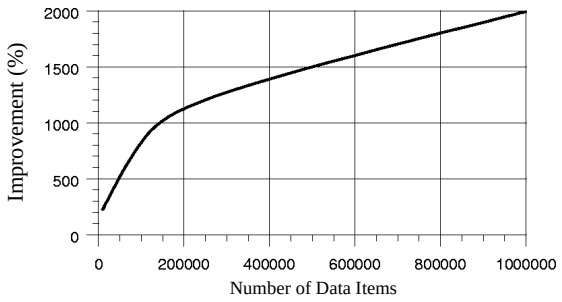
a. Improvement over Inverted Lists



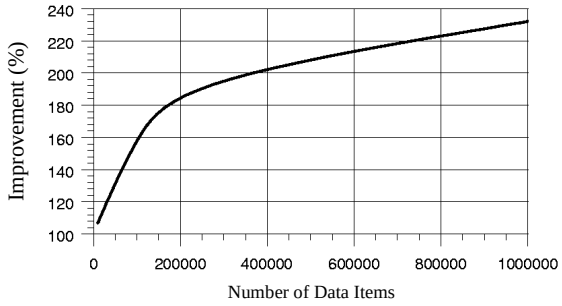
b. Improvement over Multidimensional Indexing

Figure 71: Improvement of Tree Striping for a Varying Dimension of the Data Space

tested analogously. In figure 70, we show the optimal dimensionality *(d_{opt})* of striped trees depending on the dimensionality of the data. For *d*=2 and *d*=4, the optimal dimension assignment of tree striping provides one *d*-dimensional index, i.e. it is identical to multidimensional indexing. As expected according to our theoretical analysis, for higher dimensions the optimal dimension assignment of tree striping is between the extreme cases: For *d*=12, we obtain two 6-dimensional indexes and for *d*=16, we obtain a division into 3 indexes with dimensionality (6, 5, 5). Note that in all experiments, the optimal dimension assignment estimated by our cost model exactly matches the measured optimum. For our experiments, we use the optimal dimension assignment as determined by our cost model.



a. Improvement over Inverted Lists



b. Improvement over Multidimensional Indexing

Figure 72: Improvement of Tree Striping for an Increasing Number of Data Items

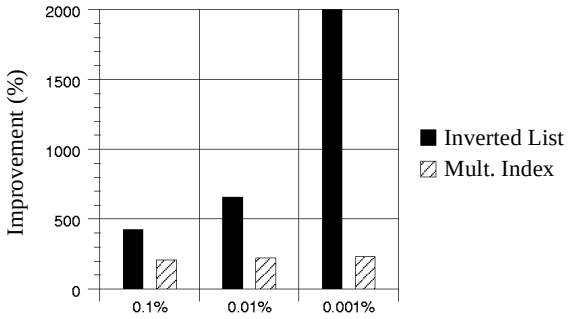


Figure 73: Performance for Varying

In the next experiment, we determined the improvement achieved by the tree striping technique. Again, we used 1,000,000 uniformly distributed data objects of varying dimensionality (d=2..16) and a query selectivity of 10^{-5}. Figure 71 depicts the results of this experiment. As expected, the improvement factor achieved over the inverted lists is much higher than the improvement over multidimensional indexing. The maximum improvement 12,300% (i.e., tree striping is 123 times faster than inverted lists) occurs for a dimensionality of 4. The tree striping technique is at least 10 times faster than the inverted lists for any experiment. The improvement over the multidimensional indexing increased with increasing dimensionality of the data space. For dimensions smaller than 8 there was no or only a negligible improvement. This means that for low dimensionality, tree striping corresponds to multidimensional indexing. For higher dimensions, however, the improvement rapidly increases and reaches about 200% for a dimensionality of 16. Thus, for high-dimensional data the tree striping technique is more than twice as fast as multidimensional indexing.

Another important criterion for the evaluation of indexing techniques is their scalability, i.e. the behavior of the technique for an increasing size of the database. Therefore, we performed an experiment using a fixed dimensionality (d=16) and a fixed query selectivity of 10^{-5} and varied the number of data items from 10,000 to 1,000,000. Again, we used our cost model to determine the optimal dimension assignment. The improvement over multidimensional indexing starts with a moderate value of 107% for a small database but, as the size of the database increases, the improvement also increases up to 230% over multidimensional indexing for the largest database of 1,000,000 objects (cf.

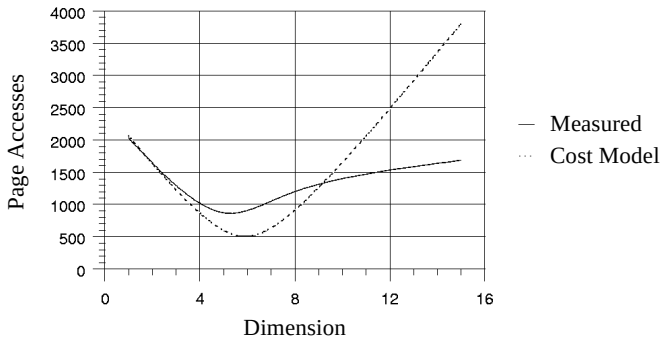


Figure 74: Optimal Dimension Assignment for Real Data (Text Data)

figure 72). The improvement over the inverted list approach starts with 228% and reaches its maximum by 2,000% (20 times faster) for the largest database of 1,000,000 objects (cf. figure 72).

The intention of the experiment depicted in figure 73 is to show that the high improvements are independent from the selectivity of the queries. We repeated the previous experiments for different dimensionality (shown are the experiment for d=12 and d=16) using selectivities between 10^{-3} and 10^{-5}. Again, we obtained an improvement factor of 210% to 220% over the multidimensional index and an improvement factor of 4 to 20 over the inverted lists.

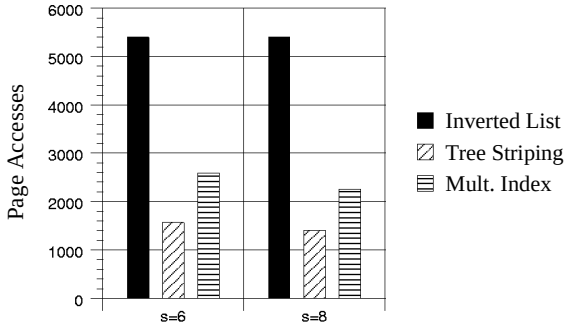


Figure 75: Performance of Partial Range Queries

To show the practical relevance of our technique, we also evaluated the performance of tree striping for other important query types. One of the most important query types is the partial range query. In our experiments with partial range queries, again we used 1,000,000 uniformly distributed objects (d=15). We randomly generated partial range queries specifying a query range on 6 attributes for the first experiment and 8 attributes for the second experiment. All queries have a selectivity of 10^{-5} leading to an average result of 10 objects. For the tree striping technique, we determined an optimal dimension assignment of three 5-dimensional indexes. The results presented in figure 75 show that the tree striping technique outperforms the inverted lists and the multidimensional index. The achieved improvement was 345% (for the partial range queries on 6 attributes) and 303% (for the partial range queries on 8 attributes) over the inverted lists, and 166% (6 attributes) and 160% (8 attributes) for the multidimensional indexing approach.

In a last series of experiments, we evaluated the tree striping technique using real data which consists of text data describing substrings of a large database of texts. In figure 74, we compare the measured performance for range queries with a selectivity of 0.2% to the performance determined by our model (cf. section 5.3). The minima of the two curves correspond to the optimal dimension assignment (d_{opt}). Note that the model estimates the optimal dimension assignment correctly ($d_{opt} \approx 5$), although it assumes a uniform distribution of the data. The difference between model and measurements for large dimensions (i.e. small k), however, may be explained by the non-uniform distribution of the real data.

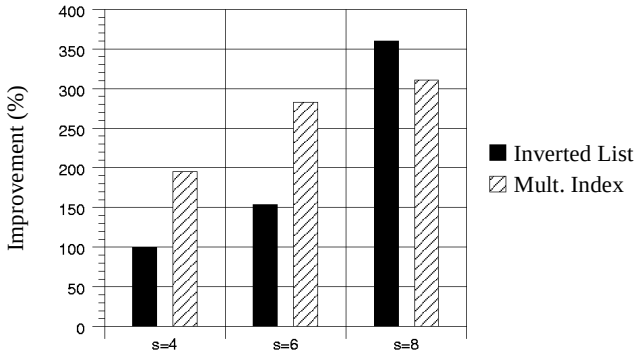


Figure 76: Improvement for Partial Range Queries

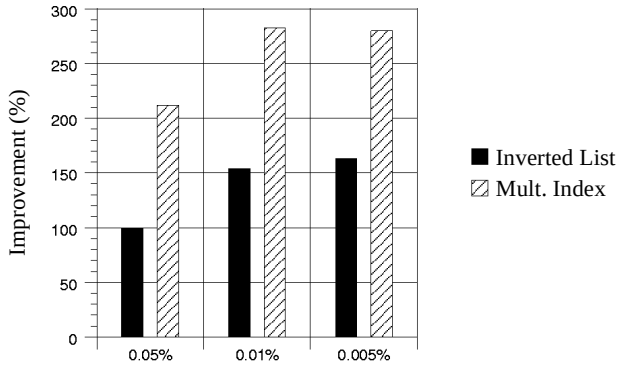


Figure 77: Performance of Partial Range Queries with Varying Selectivities (Text Data)

In figure 76, we present the improvement of tree striping over inverted lists and multidimensional indexing for partial range queries with a varying number of specified attributes (s=4..8). It is interesting that for a partial range query with 4 specified attributes, tree striping degenerates to inverted lists. If more than 4 attributes are specified, tree striping becomes better than both, inverted lists and multidimensional indexing. Note that for s=6, inverted lists are better than multidimensional indexing whereas for s=8, multidimensional indexing is better than inverted lists.

In a last experiment with real data, we varied the selectivity of the partial range queries. Figure 77 shows three different selectivities (0.05%, 0.01%, 0.005%) of partial range queries each having six attributes specified (s=6). Note that tree striping is consistently better than the inverted lists and multidimensional indexing approaches, and the improvement factor increases with a decreasing selectivity of the partial range queries.

Chapter 6
Optimizing the Geometry of Regions Using Bulk-Load Operations

In this chapter, we will exploit the potential for optimizing the shape of the bounding boxes. The classical approaches for low-dimensional query processing [BKSS 90] tend to optimize for cube-like bounding boxes forcing all side lengths to be in the same order of magnitude. This is achieved by dynamically inserting the data vectors into the data pages, splitting the data pages whenever an overflow occurs. From our model presented in chapter 3 it can be derived that optimization for cubes is appropriate in low dimensions. However, we will show in section 6.4 that this optimization leads to a deteriorated performance behavior in high-dimensional query processing. We will derive from our model that range searches in high-dimensional data spaces become more efficient when thin pages are cut from the borders of the data space. It is difficult to achieve such space partitioning in a dynamic index construction. Therefore, we describe our geometry optimization in the context of a bulk-loading technique for high-dimensional indexes.

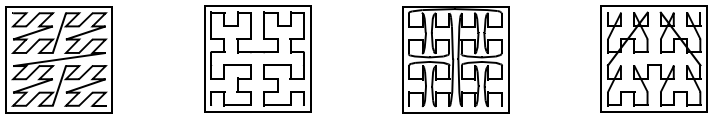
The benefit of this chapter is therefore two-fold: Additionally to the performance gain for the search operation, we present a sophisticated new algorithm for the index construction improving the efficiency of this operation by orders of magnitude. This improvement will be shown both analytically as well as experimentally. Parts of the material presented in this chapter were published [BBK 98].

6.1 Introduction

A typical database application starts with an empty database which will grow continuously by multiple insert operations. It is not appropriate to use an index structure in the beginning of this process because having only a relatively small amount of high-dimensional feature vectors, a sequential scan of the data will be much faster than an index based search. Therefore, we are supposed to simply store the feature vectors on the disk and scan the whole database for query processing. When the size of the database reaches a certain value, the use of an index structure is required. We may use a cost model as discussed in chapter 3 to determine this break-even point for a given dimensionality of the feature vectors. At this point, we face the problem to build an index file from a large amount of data, i.e. to bulk-load the index. As the process of inserting data does not stop at that time, we cannot use a static index structure but have to use a dynamic index structure and additionally supply it with an efficient bulk-load operation. As the X-tree outperforms the TV-tree and the R^*-tree regarding the search performance, we decided to use the X-tree as an index structure.

On the other hand, we may draw some advantage from the fact that we do not only know a single data item - as in case of a normal insertion operation - but a large amount of data items. It is a common knowledge that we can achieve a higher fanout and storage utilization using bulk-load operations resulting in a slightly better search performance. But do we exhaust all the potential of this information by increasing the storage utilization? As we will see later in this chapter, we do not. This is due to the fact that a priori knowing all data allows us to choose an alternate data space partitioning. As we will show analytically, space partitioning caused by a split strategy splitting the data space in two equally-sized portions performs poor in contrast to an unbalanced split. An experimental evaluation of our bulk-loading technique proves this result.

The rest of this chapter is organized as follows: In section 6.3, we introduce the general idea of bulk-loading an index structure. We then give an overview over existing bulk-loading techniques and analyze their behavior especially concerning effects occurring in high-dimensional spaces. In section 6.3.3, we theoretically analyze the performance of various split strategies in high-dimensional data spaces. In section 6.4, we propose our new technique which allows not only a fast bulk-load operation but also results in a better space partitioning than we can achieve using a dynamic index structure. In section 6.3.7, we analytically show that our bulk-load operation can be done in



Peano (Z-Order) **Hilbert** **Gray-Codes** **U-Index**

Figure 78: Space Filling Curves

O(*n* log *n*) time. The chapter is concluded by a variety of experimental results which demonstrate the advantage of our technique compared to dynamic indexing and other bulk-loading techniques.

6.2 Related Work

6.2.1 General Idea of bulk-loading

Building an index from a given large set of high-dimensional vectors, we have to divide the data set into rather small portions which fit into a single data page. Thus, we have to assign each vector of the data set to a data page and additionally build an appropriate directory. There are two known techniques to assign data items to data pages: We can use a function $\Re^d \rightarrow \Re$ which provides a one-dimensional order of the data space, and – after sorting the data items – sequentially assign them to data pages. Usually, a space filling curve such as the Hilbert curve or Z-ordering is used as an assigning function. Figure 78 shows some two-dimensional examples of space filling curves. Space filling curves can also be directly applied for multidimensional indexing, cf. chapter 2.

As an alternative, we can divide the data space into partitions which correspond to data pages. This partitioning of the data space can be done in a top-down fashion which means that we hierarchically divide the *d*-dimensional space using (*d*–1)-dimensional hyperplanes as borderlines between the partitions. More formally, we divide the *d*-dimensional space ds_0 into n_0 partitions $ds_{0\,0}...ds_{0\,n0}$. These partitions $ds_{0\,i}$ are then split into partitions $ds_{0\,i\,0}...ds_{0\,i\,n1}$ and so on. In addition, however, we have to assure that a directory can be built on top of this space partitioning, i.e. we have to meet some restrictions with respect to the values n_i.

Frequently, bulk-loading an index is also called bottom-up construction of the index. This is due to the fact that we first construct the data pages which are at the “bottom” of

the index structure and then construct the directory pages. As this term is misleading because we actually partition the data space in a top-down fashion, we omit this term and use bulk-loading or simply index construction instead.

6.2.2 Hilbert R-Trees

For both of the general techniques, some research has been done. From the class of space filling curves, also known as fractals, the Hilbert curve seems to be the most appropriate technique for multidimensional indexing. The relevant property hereby is the preservation of neighborhood which means that objects which are close in the d-dimensional space should be close in the 1-dimensional space, too. As experiments show, in case of the Hilbert curve this property holds for most of the points. This leads to the development of the Hilbert R-tree [KF 94]. A Hilbert R-tree is created by externally sorting all the data vectors according to their Hilbert value. Then, we divide the resulting sorted array of vectors into equally sized portions such that every portion fits into one data page. We store the corresponding vectors into data pages. In the next step, we divide the resulting array of data pages which is still sorted according to the Hilbert value into equally sized portions and determine the corresponding minimum bounding boxes (MBR). We finally store the MBRs in directory pages clustering these directory pages recursively until we reach a single root node. The costs for bulk-loading a Hilbert R-tree are obviously in $O(n \log n)$ time due to external sorting. The Hilbert R-tree performs very well for low-dimensional spaces. In these spaces, it outperforms the Z-order space filling curve and is competitive to dynamic R-trees. However, we are not able to predict the behavior of an index structure in high-dimensional spaces from the behavior in low-dimensional spaces. As we will see in section 6.5, the Hilbert ordering degenerates in higher dimensions leading to a bad query performance. The reason for this behavior is the resulting overlap when creating data pages from sorted Hilbert values. Obviously, a range in the 1-dimensional Hilbert space does not correspond to a rectangular region in the d-dimensional space. This introduces an overlap into the index which increases when going to higher dimensions. Additionally, we do not have the choice of adapting the space partitioning to the data distribution.

6.2.3 VAM-Split R-Trees

The VAM-Split trees which have been proposed by White and Jain use the concept of hierarchical space partitioning. VAM-Split trees are R-trees or KDB-trees which are

bulk-loaded by creating a kd-tree-like structure. The data vectors are initially stored in an array. Then, the algorithm determines a split dimension and a split value within this dimension. The value is determined such that the variance from each point to the split value in maximized. According to the authors, this can be done in O(n) time. In the next step, all data vectors are transferred to the upper or the lower half of the array depending on the value of the vector in the split dimension. From that, the algorithm has partitioned the data space into two portions which are separated by a (d-1)-dimensional hyperplane. The split dimension is the normal vector of the hyperplane. The algorithm recursively repeats the partitioning process until portions of the space exist which fit into a single data page. Except the fact that the split condition was maximizing the variance instead of using the median, this technique is very similar to building a kd-tree. The disadvantage of the algorithm is that it does neither take advantage of the knowledge that we have from the fact that the whole amount of data is present during the bulk-load operation nor effects occurring in high-dimensional spaces have been taken into account. As we will see, this results in an inadequate data space partitioning.

6.2.4 Buffer Trees

In [BSW 97], van den Bercken, Seeger and Widmayer propose a new technique called buffer trees. Buffer trees are a generalized technique which potentially works on all multidimensional index structures. The buffer tree is a derivative of the data structure to be constructed (called the ‘target’ index structure) with two major modifications: First, an additional buffer is assigned to each directory page, and second, the capacity of a directory page may differ from the capacity of the target data structure. The buffer of each directory page is partially held in the main memory and partially laid out on the secondary storage. During the bulk-load operation, each tuple is inserted into the buffer of the root node. If the buffer of the root node is full, all objects in the buffer are dispatched to the next deeper index level. This process continues until the data level is reached. If the last object has been inserted into the buffer tree, all buffers are emptied by propagating the contained points down the tree. The data pages of the buffer tree can be taken as data pages of the target index while the directory of the buffer tree has to be discarded due to incorrect capacity. The various directory levels of the target index are created by inserting the bounding boxes into further buffer trees. Number, capacity and buffer size of the directory nodes are limited by the available main memory and have to be optimized accordingly. Although avoiding a complete sorting of the data set, the

authors prove that the lower bound of page accesses in external sorting is achieved by their algorithm. A significant performance improvement over dynamic index construction is shown experimentally for R-trees. The general advantage of the buffer tree approach is that, algorithms designed for tuning the query performance of the target index structure can be applied without modification. Obviously, the resulting index has the same properties as a dynamically constructed index. On the other hand, no specific advantage is taken from knowing the complete data set a priori. Additionally, an overlap in the target directory is not avoided.

6.3 Our New Technique

In this section, we present our new bulk-loading technique. Although applicable to most R-tree-like index structures, we decided to use the X-tree as an example because according to [BKK 96], the X-tree outperforms other high-dimensional index structures. In contrast to dynamically constructed X-trees, our algorithm exploits a priori knowledge of the complete data set to create an overlap-free directory, also avoiding supernodes. An arbitrary storage utilization can be achieved, including a near-100% utilization. As we will see later, “near 100%” means 100% up to round-off effects. Furthermore, if we choose a storage utilization lower than 100%, we use the gained freedom for an acceleration of the construction.

6.3.1 Basic Idea

During the bulk-load operation, the complete data is held on the secondary storage. Although only a small cache in the main memory is required, cost intensive disk operations such as random seeks are minimized. In our algorithms, we strictly separated the split strategy from the core of the construction algorithm. Therefore, we can easily replace the split strategy and thus, create an arbitrary overlap-free partition with the given storage utilization. Various criteria for the choice of direction and position of split hyperplanes can be applied. Especially, we have implemented various kinds of asymmetric split strategies which are not applicable in a dynamic index construction.

The index construction is a recursive algorithm containing the following subtasks:

- determining the tree topology (height, fanout of the directory nodes, etc.)
- the split strategy,

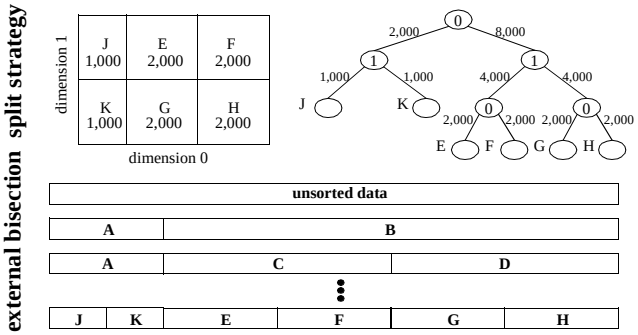


Figure 79: Basic Idea of Our Technique

- external bisection of the data set according to tree topology and split strategy
- construction of the index directory.

Although all these subtasks run in a nested fashion, we will present them separately to maintain clarity. However, we cannot isolate the split strategy and the bipartitioning algorithm from the tree topology. For example, in order to achieve an overlap-free directory, both the split strategy and the partitioning algorithm have to consider the fanout of an individual directory node as provided by the tree topology (Note that, at least in the highest levels of the tree, the fanout can be much smaller than the page capacity, as we will show later). Vice versa, the exact topology of a subtree can only be determined if the exact number of data objects stored in this subtree is known. This exact number, however, depends on the results of previous splits and bisections. Thus, although we separately describe the parts of the algorithms, we have to keep in mind the special requirements and prerequisites of the other parts.

An example will clarify the idea of our algorithm: Let us assume that we have given 10,000 two-dimensional data items and we can take from several properties our index structure that 10,000 items will fill a tree of height 3 having 6 entries in the root node (determination of tree topology). Thus, we first call the recursive partitioning algorithm which applies the split strategy to our 10,000 data items and gets the following back:

“The 10,000 items should be first split according to dimension 0 such that partition A contains 2,000 items and partition B contains 8,000 items. Then we should split partition B according to dimension 1 such that partition C and D each contain 4,000 items. Again,

we should split C and D according to dimension 0 that each of the partitions E, F, G, and H each contain 2,000. Finally, we should split partition A according to dimension 1 into partitions J and K such that J and K contain each 1,000 data items."

Note that this information could also be seen as a binary tree (split tree) having split dimensions as nodes and amounts of data as denotations of edges. The upper part of figure 79 depicts the result of the split strategy and the corresponding split tree. As next step, the top-down partitioning algorithm calls the external bisection algorithm ("External" means that the data to be bisected is located on the secondary storage and the algorithm also operates on disk) which divides the previously unsorted data into the six desired portions (E, F, ..., J, K). This is depicted in the lower part of figure 79. At this point, we have partitioned our root node into the six subtrees. Note that the data inside the partitions (J, K, ..., G, H) remains unsorted during the bisection, i.e. there exists no ordering inside of J. As last step, we recursively apply our algorithm to the six partitions until we reach the data pages and write the corresponding directory to the secondary storage.

6.3.2 Determination of the Tree Topology

The first prerequisite of our algorithm is to determine the topology of the tree resulting from our bulk-load operation. The topology of a tree includes the height of the tree, the fanout of the directory nodes on the various tree levels, the capacity of data pages, and the number of objects stored in each subtree. However, we do not regard the exact number of objects stored in a tree, but a range between a maximum and a minimum number. The topology of the tree only depends on static information which is invariant during the construction such as the number of objects, the dimension of the data space, the page capacity and the storage utilization.

Let $C_{\text{max,data}}$ be the maximum number of data objects in a data page where

$$C_{\text{max, data}} = \left\lfloor \frac{\text{pagesize}}{\text{sizeof(dataobject)}} \right\rfloor,$$

$C_{\text{max,dir}}$ analogously the maximum fanout of a directory page, and $C_{\text{eff,data}}$ and $C_{\text{eff,dir}}$ the average capacity of a data/directory page with

$$C_{\text{eff,data}} = \text{storageutilization} \cdot C_{\text{max, data}}.$$

The maximum number of data objects stored in a tree with height *h* is then:

$$C_{\max,\text{tree}}(h) = C_{\max,\text{data}} \cdot C_{\max,\text{dir}}^{h-1} \qquad C_{\text{eff},\text{tree}}(h) = C_{\text{eff},\text{data}} \cdot C_{\text{eff},\text{dir}}^{h-1}.$$

Therefore, the height of the tree must initially be determined such that $C_{\text{eff,tree}}$ is greater than the actual number of objects *n*. More formally:

$$h = \left\lceil \log_{C_{\text{eff,dir}}}\left(\frac{n}{C_{\text{eff,data}}}\right) \right\rceil + 1$$

Note that we have to evaluate this formula only once in order to determine the level of the root node of the index. As the X-tree and other R-tree related index structures are always height-balanced, we can easily determine the level of subtrees by decrementing the level of the parent node of the subtree. Now, we have to determine the fanout of the root node of a tree *T* with height *h* when filled with *n* data objects. Let us assume that every subtree of height *(h-1)* is filled according to its average capacity $C_{\text{eff,tree}}(h-1)$. Thus, the fanout is the quotient of *n* and the average capacity of the subtrees:

$$\text{fanout}(h,n) = \min\left(\left\lceil \frac{n}{C_{\text{eff,tree}}(h-1)} \right\rceil, C_{\max,\text{dir}}\right) = \min\left(\left\lceil \frac{n}{C_{\text{eff,data}} \cdot C_{\text{eff,dir}}^{h-2}} \right\rceil, C_{\max,\text{dir}}\right).$$

The minimum is required due to round-off effects.

Obviously, a 100% storage utilization in every node can be achieved only for certain values of *n*. Usually, the number of nodes in each level must be rounded-up. Thus, the data nodes and their parents are utilized best according to the desired storage utilization while the worst utilization typically occurs in the top levels of the tree. In general, our algorithm creates the highest possible average storage utilization below the chosen one.

6.3.3 The Split Strategy

Once we laid down the fanout *f* of a specific directory page *P*, the split strategy has to be applied to determine *f* subsets of the current data. As we regard the split strategy as an replaceable part of our algorithm, we only describe the requirements of a split strategy in this section. A detailed description of our optimized split strategy will be given in section 6.4.

Assuming that the data set is bisected repeatedly, the split strategy determines the binary *split tree* for a directory page which has *f* leaf nodes and may be arbitrarily unbalanced. Each non-leaf node in the split tree represents a hyperplane (the split plane) splitting the data set into two subsets. The split plane can be described by the split dimension

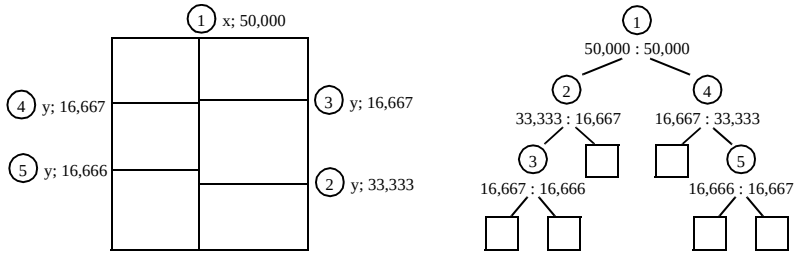


Figure 80: The Split Tree

and the numbers of data objects (NDO) on each side of the split plane. Thus, a split strategy has to determine the split dimension and the ratio between the two NDOs. Furthermore, we allow the split strategy to produce not only constant ratio but a interval of acceptable ratios. We will use this freedom later to accelerate the bisection algorithm. Note that the split strategy does not provide the position of the split plane in terms of attribute values. We determine this position using the bisection algorithm.

In every subtree of the split tree, the number of data objects NDO is proportional to the number of leaf nodes in the split tree:

$$\text{NDO} = n \cdot \frac{\#\text{leafnodes}}{f}$$

Figure 80 shows an example of a subtree containing 100,000 data objects to be organized in a directory node of fanout 6. When applying a symmetric split strategy, the first split has to be in the middle of the data set dividing it into two subsets with NDO_1=50,000 elements according to the x-axis. Then, we have to divide each of the subsets into 3 parts. This is done by a 2:1 split into NDO_2=33,333 and NDO_3=16,667 objects according to the y-axis (node 2). The rectangle containing 33,333 elements is then again partitioned into two approximately equally sized subsets. An alternate split strategy, for example, cuts NDO_1=16,666 objects out of the 100,000 in the first step, then again NDO_2=16,666 from the rest, and so on. In this case, the split tree degenerates to a linear list.

In order to determine the split dimension, we have to consider two cases: If the data subset fits into the main memory, the split strategy can determine the split dimension and the subset size by computing selectivities or variances from the complete data subset.

Otherwise, decisions are based on a sample of the subset which fits into the main memory and can be loaded without causing too many random seek operations. We use a simple heuristic to sample the data subset which loads subsequent blocks from three different places in the data set.

6.3.4 Recursive Top-Down Partitioning

Now, we are able to define a recursive algorithm for partitioning the data set. The algorithm consists of two procedures which are nested recursively (both procedures call one another). The first, *partition()*, is called once for each directory page. Its duties are:

- call the topology module to determine the fanout of the current directory page
- call the split-strategy module to determine a split tree for the current directory page
- call the second procedure, *partition_acc_to_split_tree()*

The second function partitions the data set according to the split dimensions and the proportions given in the split tree. However, the proportions are not regarded as fixed values. Instead, we will determine lower and upper bounds for the number of objects on each side of the split plane. This will help us to improve the performance of the next step, external bipartitioning. Let us assume that the number of leaf nodes on each side of the current node in the split tree is $l : r$, and that we are currently dealing with N data objects. An exact split plane would exploit the proportions

$$N_{\text{left}} = N \cdot \frac{l}{l+r} \text{ and } N_{\text{right}} = N \cdot \frac{r}{l+r} = N - N_{\text{left}}.$$

Instead of using the exact values, we compute an upper limit for N_{left} such that N_{left} is not too large to be placed in l subtrees with height $h-1$ and a lower limit for N_{left} such that N_{right} is not too large for r subtrees:

$$N_{\text{max,left}} = l \cdot C_{\text{max,tree}}(h-1) \qquad N_{\text{min,left}} = N - N_{\text{max,right}} = N - r \cdot C_{\text{max,tree}}(h-1)$$

An overview over the algorithm is depicted in C-like pseudocode in figure 81. For the presentation of the algorithm, we assume that the data vectors are stored in an array on the secondary storage and the current data subset is referred to by the parameters start and n, where n is the number of data objects and start represents the address of the first object.

The procedure *index_construction(n)* determines the height of the tree and calls *partition()* which is responsible for the generation of a complete data or directory page. The details of the page generation are provided in section 6.3.6. The function *partition()* first

```
index_construction (int n)
{
      int h = (int)(log (n/Ceffdata) / log (Ceffdir) + 1) ;
      partition (0, n, h) ;
}

partition (int start, int n, int height)
{
      if (height == 0) {
            ... // write data page, propagate info to parent
            return ;
      }
      int f = fanout (height, n) ;
      SplitTree st = split_strategy (start, n, f) ;
      partition_acc_to_splittree (start, n, height, st) ;
      ... // write directory page, propagate info to parent
}

partition_acc_to_splittree (int start, int n, int height,
                            SplitTree st)
{
      if (is_leaf (st)) {
            partition (start, n, height - 1) ;
            return ;
      }
      int mtc = max_tree_capacity (height - 1) ;
      n_maxleft = st->l_leaves * mtc ;
      n_minleft = N - st->r_leaves * mtc ;
      n_real = external_bipartition (start, n, st->splitdim,
                                     n_minleft, n_maxleft) ;
      partition_acc_to_splittree (start, n_real,
                                  st->leftchild, height) ;
      partition_acc_to_splittree (start + n_real, n - n_real,
                                  st->rightchild, height) ;
}
```

Figure 81: Recursive Top-Down Data Set Partitioning

determines the fanout of the current page and calls *split_strategy()* to construct an adequate split tree. It then calls *partition_acc_to_splittree*() to get the data set partitioned according to the split tree. After partitioning the data, *partition_acc_to_splittree*() calls *partition()*, in order to create the next deeper index level. The height of the current subtree is decremented in this indirect recursive call. Therefore, the data set is partitioned in a top-down manner, i.e. the data set is first partitioned with respect to the highest directory level below the root node.

6.3.5 External Bipartitioning of the Data Set

Our bipartitioning algorithm is comparable to the well-known Quicksort algorithm [Hoa 62, Sed 78]. Bipartitioning means to split the data set or a subset thereof into two portions according to the value of one specific dimension, the split dimension. After the bipartitioning, the "lower" part of the data set contains values in the split dimension which are lower than a threshold value (the split value), the values in the "higher" part will be higher than the split value. The split value is initially unknown and is determined during the run of the bipartitioning algorithm. Note that the actual goal of the bipartitioning algorithm is to divide the array such that a specific proportion in the number of objects results where the term "proportion" is fuzzily defined as an interval. For example, we may have an array of 10,000 data vectors which we want to bipartition such that one partition contains between 3,000 and 3,500 data vectors whereas the other partition contains the rest of the data vectors, i.e. between 6,500 and 7,000 data vectors. As we do not know a priori which values are located in the interval from object 3,000 to object 3,500, we do not initially know the split value.

Bipartitioning is closely related to sorting the data set according to the split dimension. In fact, if the data is already sorted, bipartitioning of any proportion can easily be achieved by cutting the sorted data set into two subsets. However, sorting has a complexity of o(n log n), and a complete sort-order is not required for our purpose. Instead, we will present a bipartitioning algorithm with an average-case complexity of O(n). The basic idea of our algorithm is to adapt Quicksort as follows: Quicksort makes a bisection of the data according to a heuristically chosen pivot value and then recursively calls Quicksort for both subsets. Our first modification is to make only one recursive call for the subset which contains the split interval. We are able to do that because the objects on the other subsets are on the correct side of the split interval anyway and need no further sorting. Figure 82 depicts this modification. In the example, there is no need to continue sorting on the left partition (2, 1) because all elements in the left partition are already below the final split interval. The second modification is to stop the recursion if the position of the pivot value is inside the split interval (inside the grey area in figure 82). The third modification is to choose the pivot values according to the proportion rather than trying to reach the middle.

Our bipartitioning algorithm works on the secondary storage. It is well-known that the Mergesort algorithm is better suited for external sorting than Quicksort. However, Mergesort does not facilitate our modifications leading to an O(n) complexity and was

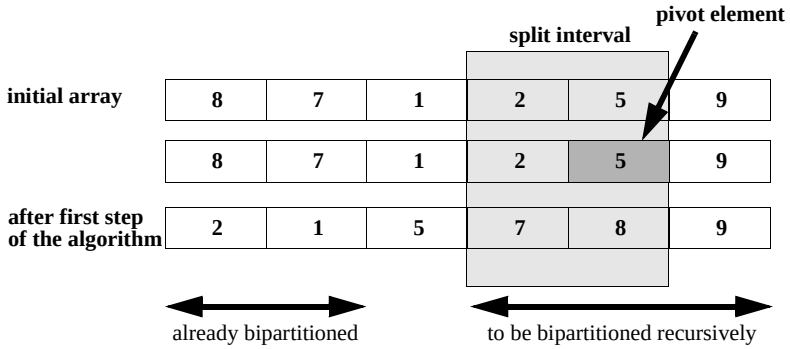


Figure 82: Adapted Quicksort

not further investigated for this reason. In our implementation, we use a sophisticated scheme reducing disk I/O and especially random seek operations much more than a normal caching algorithm would be able to.

The algorithm runs in two modes: in an internal mode if the data set to be partitioned fits in the main memory cache, and in an external mode if it does not. The internal mode is quite similar to Quicksort: The middle of three split attribute values in the database is taken as pivot value. The first object in the left side having a split attribute value larger than the pivot value is exchanged with the last element in the right side lower than the pivot value until left and right object pointers meet at the bisection point. The algorithm stops if the bisection point is inside the goal interval. Otherwise, the algorithm continues recursively with the data subset containing the goal interval.

The external mode is more sophisticated: First, the pivot value is determined from the sample which is taken in the same way as described in section 6.3.3 and can often be reused. A complete internal bipartition runs on the sample data set to determine the pivot value as well as possible. In the following external bisection (cf. figure 83), transfers from and to the cache are always processed with a blocksize half of the cache size. The cache, however, does not exactly represent two blocks on disk. Figure 83a shows the initialization of the cache from the first and last block in the disk file. Then the data in the cache is processed by internal bisection with respect to the pivot value. If the bisection point is in the lower part of the cache (figure 83c), the right side contains more objects than fit in one block. One block, starting from the bisection point, is written back

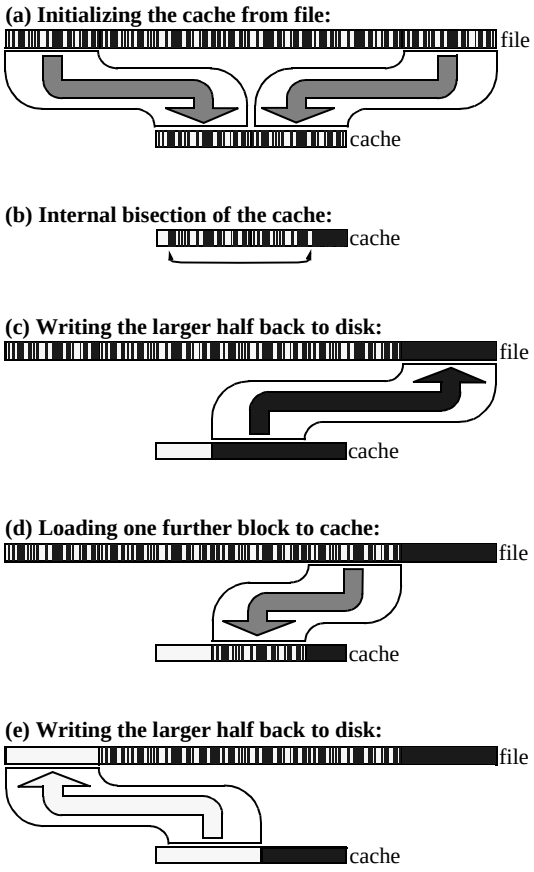


Figure 83: External Bisection

to the file and the next block is read and internally bisected again. Usually, objects remain on the lower and higher ends of the cache. These objects are used later to make transfer blocks complete. All remaining data is written back in the very last step in the middle of the file where additionally a fraction of a block has to be processed. Finally, we test if the bisection point of the external bisection is in the split interval. If the point is outside, another recursion is required.

6.3.6 Constructing the Index Directory

As the data partitioning is done by a recursive algorithm, the structure of the index is represented by the recursion tree. Therefore, we are able to create a directory node after the completion of the recursive calls for the child nodes. These recursive calls return the bounding boxes and the corresponding secondary storage addresses to the caller, where the informations are collected. There, the directory node is written, the bounding boxes are combined to a single bounding box comprising of all boxes of child nodes, and the result is again propagated to the next higher level.

Thus, a depth-first post-order sequentialization of the index is written to disk. The sequentialization starts with a sequence of data pages, followed by the directory page which is the common parent of these data pages. A sequence of such blocks is followed by a second-level directory page, and so on. The root page of the directory is the last page in the index file. As geometrically neighboring data pages are also likely to be in the same hierarchical branch, they are well clustered.

6.3.7 Analytical Evaluation of the Construction Algorithm

In this section, we will show that our bottom-up construction algorithm has an average complexity of the order O(n log n). Moreover, we will regard disk accesses in a more exact way, and thus provide an analytically derived improvement factor over the dynamic index construction. For the file I/O, we determine two measure numbers: The number of random seek operations and the amount of data read or written from or to disk. Unless no further caching is performed (which is true for our application, but cannot be guaranteed for the operating system) and provided that seeks are really random, the I/O processing time can be determined as

$$t_{\text{i/o}} = t_{\text{seek}} \cdot \text{seek\_ops} + t_{\text{transfer}} \cdot \text{amount}\,.$$

In the following, we denote by the cache capacity C_{cache} the number of objects fitting in the cache:

$$C_{\text{cache}} = \frac{\text{cachesize}}{\text{sizeof (object)}}$$

Lemma 7: Complexity of *bisection*

The bisection algorithm has the complexity O(n).

Proof (Lemma 7)

We assume that the pivot element is randomly chosen from the data set. After the first run of the algorithm, the pivot element is located with uniform probability at one of the n positions in the file. Therefore, the next run of the algorithm will have the length k with a probability $1/n$ for each $1<k<n$. Therefore, the cost function $C(n)$ encompasses the cost for the algorithm, $n+1$ comparison operations plus a probability weighted sum of the cost for processing the algorithm with length $k-1$, $C(k)$. We get the following recursive equation:

$$C(n) = n+1+\sum_{k=1}^{n}\frac{C(k-1)}{n},$$

which can be solved by multiplying with n and subtracting the same equation for $n-1$:

$$n\cdot C(n)-(n-1)\cdot C(n-1) = n\cdot(n+1)-n\cdot(n-1)+\sum_{k=1}^{n}C(k-1)-\sum_{k=1}^{n-1}C(k-1)$$

This can be simplified to

$$C(n) = 2+C(n-1),$$

and, as $C(1) = 1$,

$$C(n) = 2\cdot n = \mathrm{O}(n),$$

❑

Lemma 8: Cost Bounds of Recursion

(1) The amount of data read or written during one recursion of our technique does not exceed four times the file-size.

(2) The number of seek operations required is bounded by

$$\text{seek\_ops}(n)\le\frac{8\cdot n}{C_{\text{cache}}}+2\cdot\log_2(n)$$

Proof (Lemma 8)

(1) follows directly from Lemma 1 because every compared element has to be transferred at most once from disk to main memory and at most once back to disk.

(2) In each run of the external bisection algorithm, file I/O is processed with a block-size of cachesize/2. The number of blocks read in each run is therefore

$$\text{blocks\_read}_{\text{bisection}}(n) = \frac{n}{C_{\text{cache}}/2} + 1$$

because one extra read is required in the final step. The number of write operations is the same such that

$$\text{seek\_ops}(n) = 2 \cdot \sum_{i=0}^{r_{\text{interval}}} \text{blocks\_read}_{\text{run}}(i) \leq \frac{8 \cdot n}{C_{\text{cache}}} + 2 \cdot \log_2(n),$$

❑

Lemma 9: Average Case Complexity of Our Technique

Our technique has an average case complexity O(n log n) unless the split strategy has a complexity worse than O(n).

Proof (Lemma 9)

For each level of the tree, the complete data set has to be bisectioned as often as the height of the split tree. As the height of the split tree is limited by the directory page capacity, there are at most

$$h(n) \cdot C_{\text{max,dir}} = \text{O}(\log n)$$

bisection runs necessary. Our technique has therefore the complexity O(n log n),

❑

Lemma 10: Cost of Symmetric Partitioning

For symmetric splitting, the *partition()* procedure has an amount of file I/O data of

$$\left(\log_2(\frac{n}{C_{\text{cache}}}) + \log_{C_{\text{max,dir}}}(\frac{n}{C_{\text{cache}}})\right) \cdot 4 \cdot \text{filesize}$$

and requires

$$\left(\log_2(\frac{n}{C_{\text{cache}}}) + \log_{C_{\text{max,dir}}}(\frac{n}{C_{\text{cache}}})\right) \cdot \left(\frac{8 \cdot n}{C_{\text{cache}}} + 2 \cdot \log_2(n)\right)$$

random seek operations.

Proof (Lemma 10)

The height of the cumulated partition tree (i.e. the binary tree which is formed by replacing all directory nodes of the index by the corresponding split trees) does not exceed the following bound:

$$h_{\text{cum}}(n) \leq \log_2(n) + \log_{C_{\text{max,dir}}}(n)$$

Basically, the cumulated split tree is a complete binary tree with the exception that the last split tree level in each index level is incomplete. Therefore, the height of the cumulated split tree increases in each index level at most by one compared to the complete binary tree. In the lowest levels of the cumulated split tree, no I/O transfers are necessary, because the corresponding subsets fit into the cache. These levels are not considered. The number of levels, where I/O processing is necessary, is bounded by:

$$\begin{aligned} h_{\text{processed}}(n) &\leq \log_2(n) + \log_{C_{\text{max,dir}}}(n) - \log_2(C_{\text{cache}}) - \log_{C_{\text{max,dir}}}(C_{\text{cache}}) \\ &= \log_2(\frac{n}{C_{\text{cache}}}) + \log_{C_{\text{max,dir}}}(\frac{n}{C_{\text{cache}}}) \end{aligned}$$

For each level of the cumulated split tree, the complete file is at most once completely processed. In combination with lemma 7, the formulas are proven.

❑

Lemma 11: Cost of Dynamic Index Construction

Dynamic X-tree construction requires $2\,n$ seek operations. The transferred amount of data is $2 \cdot n \cdot \text{pagesize}$.

Proof (Lemma 11)

For the X-tree, it is generally assumed that the directory is completely held in the main memory. Data pages are not cached at all. For each insert, the corresponding data page has to be loaded and written back after completing the operation.

❑

Moreover, no better caching strategy for data pages can be applied, since without preprocessing of the input data set, no locality can be exploited to establish a working set of pages. From the results of lemmata 10 and 11 we can derive an estimate for the improve-

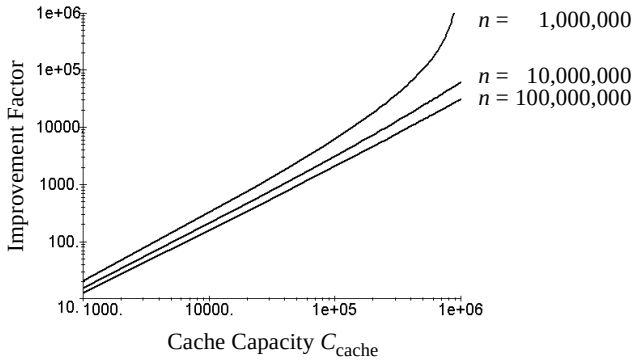


Figure 84: Improvement Factor for the Index Construction According to Lemma 7-11

ment factor of the bottom-up construction over dynamic index construction. The improvement factor for the number of seek operations is approximately:

$$\text{Improvement} \approx \frac{C_{\text{cache}}}{4 \cdot \left(\log_2(\frac{n}{C_{\text{cache}}}) + \log_{C_{\max,\text{dir}}}(\frac{n}{C_{\text{cache}}}) \right)}$$

It is almost (up to the logarithmic factor in the denominator) linear in the cache capacity. Figure 84 depicts the improvement factor (number of random seek operations) for varying cache sizes and varying database sizes.

6.4 Improving the Query Performance

In the dynamic index construction, the most important decision in split processing is the choice of the split axis whereas the split value is rather limited. Heavily unbalanced splits, such as a 10:1 proportion are commonly regarded as undesired because storage utilization guarantees would become impossible if pages with deliberately low filling degree are generated in an uncontrolled manner. Moreover, for low-dimensional spaces, it is beneficial to minimize the perimeter of the bounding boxes, i.e. to shape the bounding boxes such that all sides have approximately the same length [BKSS 90]. But, there

are some effects in high-dimensional data spaces leading to performance deterioration when minimizing the perimeter.

The first observation is that at least when applying balanced partitioning on a uniformly distributed data set, the data space cannot be split in each dimension. Assuming for example a 20-dimensional data space which has been split exactly once in each dimension would require 2^{20} = 1,000,000 data pages or 30,000,000 objects if the effective page capacity is 30 objects. Therefore, the data space is usually split once in a number d' of dimensions. In the remaining (d - d') dimensions it has not been split and the bounding boxes include almost the whole data space in these dimensions. As we assume the d-dimensional unit hypercube as data space, the bounding boxes have approximately side length 1/2 in d' dimensions and approximately side length 1 in (d - d') dimensions. The maximum split dimension d' can be determined from the number N of objects stored in the database:

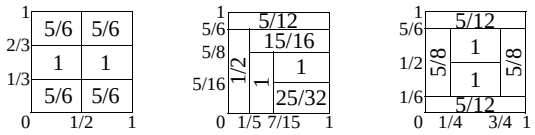
$$d' = \log_2(\frac{N}{C_{eff}}).$$

The second observation is that a similar property holds for typical range queries. If we assume that the range query is a hypercube and should have a selectivity s, then the side length q is the d^{th} root of s: $q = \sqrt[d]{s}$. For a 20-dimensional range query with selectivity 0.01% we get a side length q = 0.63 which is larger than half of the extension of the data space in this direction.

It becomes intuitively clear that a query with side length larger than 1/2 must intersect with every bounding box having at least side length 0.5 in each dimension. However, we are also able to model this effect more accurate: We adapt our model to take window queries into account. Our first modification for window queries is that we assume always the window to be entirely contained in the data space. In contrast to range queries, the event space, from which the query anchor is taken, is not the complete data space but rather a subspace.

$$P_{\text{bound\_eff}}(q) = \sum_i \prod_{0 \le j < d} \frac{\min(ub_{i,j}, 1-q) - \max(lb_{i,j} - q, 0)}{1-q}$$

The minimum and maximum is required to cut the parts of the Minkowski sum exceeding the data space. The denominator (1 - q) is required because the stochastic "event space" of the query anchor is not [0 ... 1] but rather [0 ... 1-q]. As an example, the results of three different sets of partitions for 6 pages in 2-d space and their expected page



$P_1(0.6) = 5.33$ $P_2(0.6) = 4.64$ $P_3(0.6) = 4.08$

Figure 85: Examples for Balanced and Unbalanced Split Strategies in 2-d Space

accesses for a range query with side length 0.6 are illustrated in figure 85. All bounding boxes have an area of 1/6. The individual access probability is depicted inside the boxes. The first partitioning corresponds to a balanced split strategy optimized for square-like bounding boxes. The second corresponds to a strategy cutting a slice with area 1/6 from the lower part of the remaining space. The dimensions are in this case changed periodically. The third strategy is similar to the second with the only exception that slices are cut from the lower and the higher end before the dimensions are changed. We can take from this simple 2-dimensional example that for large queries the performance is slightly (30%) improved if the pages are split unbalanced. This is due to the fact that close to the border of the data space, there arise long pages with a low access probability.

Thus, we implemented the following unbalanced split strategy: If the current data set fits into main memory, then we determine the dimension d_s where the space partition to be split has maximal extension. Otherwise, we apply the same criterion to a sample of the current data set which is taken as mentioned in section 6.3.5. Once d_s has been determined, we split the space according to the given ratio. Then, we split the larger partition on the opposite side using the same ratio and split dimension. Thus, we have symmetrically split the space into three portions: A large partition in the middle of the space and two equally sized small partitions at the border of the space. If the remaining large partition contains more elements than the capacity of a subtree is, we again choose an appropriate split dimension for the remaining partition and split it according to the given ratio. This process continues until the size of the remaining partition is below the capacity of a subtree. Note that we do not have the full freedom of splitting anywhere in the last step of this process, unless we produce underfilled pages.

6.5 Experimental Evaluation

To show the practical relevance of our bottom-up construction algorithm and of our techniques for unbalanced splitting, we have performed an extensive experimental evaluation by comparing the following index construction techniques:

- Dynamic index construction by repeatedly inserting objects
- Hilbert-R-tree construction by sorting the objects according to their Hilbert values
- our bottom-up construction method using
  - balanced (1:1) splitting
  - moderately balanced (3:1)
  - heavily (9:1) unbalanced splits.

All experiments have been computed on HP9000/780 workstations with several GBytes of secondary storage. Although our technique is applicable to most R-tree-like index structures, we decided to use the X-tree as an underlying index structure because according to [BKK 96], the X-tree outperforms other high-dimensional index structures. All programs have been implemented object-oriented in C++.

Our experimental evaluation encompasses both real and synthetic data. Our real data set consists of text data, describing substrings from a large text database. We converted the text descriptors to 300,000 points in a 16-dimensional data space (19 MBytes of raw data). The synthetic data set consists of two million uniformly distributed points normalized in the 16-dimensional unit hypercube. The synthetic raw data has a file size of 128 MBytes. We created various index files using subsets of the original data sets and projecting the data space to a lower dimensional space by omitting some of the attributes. The page size used for all indexes was 4,096 Bytes. The total amount of disk space occupied by the created indexes is about 2.8 GBytes. The index construction time for all our experiments sums up to several weeks.

In our first experiment, we compared the construction times for various indexes. The external sorting procedure of our construction method was allowed to use only a relatively small cache (32 kBytes). Note that, although our implementation does not provide any further disk I/O caching, this cannot be guaranteed for the operating system. In most experiments (unless otherwise mentioned) the storage utilization was 80%. In contrast, the Hilbert construction method was implemented in combination with internal sorting for simplicity. The construction time of the Hilbert method is therefore underestimated

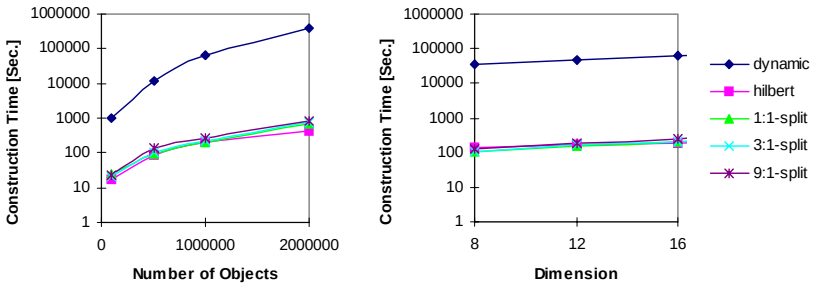


Figure 86: Performance of Index Construction Against Database Size and Dimension

by far and would worsen in combination with external sorting when the cache size is strictly limited. All Hilbert-constructed indexes have a storage utilization near 100%.

Figure 86 shows the construction time of dynamic index construction and the bottom-up methods. In the left diagram, we fixed the dimension to 16, and varied the database size from 100,000 to 2,000,000 objects of synthetic data. The resulting speed-up of the bulk-loading techniques over the dynamic construction was so enormous that we decided to use a logarithmic scale. In contrast, the bottom-up methods differ only slightly in performance. The Hilbert technique was the best method having a construction time between 17 and 429 sec. The construction time of symmetric splitting ranges from 26 to 668 sec., whereas unbalanced splitting required between 21 and 744 sec. in the moderate case and between 23 and 858 sec for the 9:1 split. In contrast, the dynamic construction time ranged from 965 to 393,310 sec (4 days, 13 hours). The improvement factor of our methods constantly increases with growing index size, starting from 37 to 45 for 100,000 objects and reaching 458 to 588 for 2,000,000 objects. The Hilbert construction is up to 915 times faster than dynamic index construction. This enormous factor is not only due to internal sorting but also to reduced overhead in changing the ordering attribute. In contrast to Hilbert construction, our technique changes the sorting criterion during the sort process according to the split tree. The more often the sorting criterion is changed, the more unbalanced the split becomes because the height of the split tree increases. Therefore, the 9:1-split has the worst improvement factor. But as we will see later, slightly higher index construction cost are amortized by far because the Hilbert construction method is completely out of the question for its poor query performance.

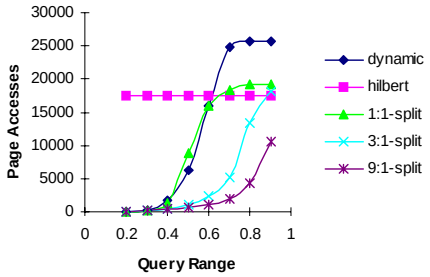


Figure 87: Performance of Range Queries with Varying Side Length

The right diagram in figure 86 shows the construction time with varying index dimension. Here, the database size was fixed to 1,000,000 objects. It can be seen that the improvement factors of the construction methods (between 240 and 320) are rather independent from the dimension of the data space.

In the next series of experiments, we determined the query performance of the various indexes using varying selectivities on synthetic data. As a query type, we used region queries because region queries serve as a basis for more complex queries such as nearest neighbor queries and, therefore, are fundamental in multimedia databases. Figure 87 shows the performance of a 16-dimensional index filled with 1,000,000 objects, uniformly distributed in the unit hypercube. We varied the selectivity of the query from $6.55 \cdot 10^{-13}$ % to 18.5%, corresponding to an edge length of the query hypercube varying

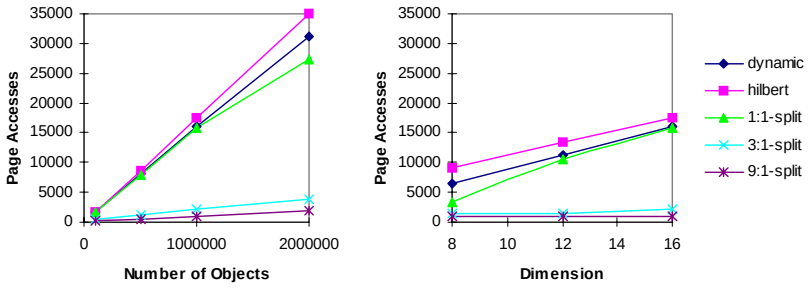


Figure 88: Performance of Range Queries with Varying Database Size and Dimension

from 0.2 to 0.9. First, we determined the number of page accesses because, for range query evaluation, disk I/O is the dominant cost factor. The result of this experiment is that the Hilbert constructed index has an unsatisfactory performance and therefore is unsuitable for indexing high-dimensional data spaces. Even very small query windows revealed a full scan of the complete index. However, dynamically and bottom-up constructed indexes with balanced splits had a very similar performance. Due to the sophisticated split strategy of the X-tree, the overlap-free directory of the bottom-up constructed index does not lead to significant performance improvements. The high storage utilization factor of the Hilbert-constructed X-tree leaded to the astonishing result that its performance is better than the performance of the dynamic index for very large queries (having an edge length greater than 0.6). The reason simply is that this index is smaller. The disadvantage of a 100% storage utilization is that performance deteriorates whenever new data has to be inserted dynamically after index construction. Although the balanced bottom-up constructed indexes avoid this disadvantage by choosing a high but not exact 100% utilization, this effect is responsible for a better efficiency for query ranges between 0.7 and 0.9.

The benefits of unbalanced splitting can be observed at any query window size. Especially the heavily unbalanced split leads to an index yielding very good performance also on very large queries. The improvement factor over the balanced split reaches 15.6 at a query edge length of 0.6. It is more than 15.7 times faster than the dynamically constructed index.

In figure 88, we confirmed these results for varying database sizes and for varying dimensions of the data space. In these experiments, we choose a selectivity of about

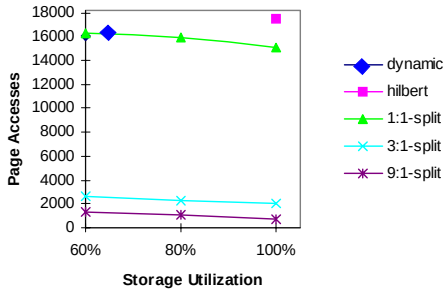


Figure 89: Influence of the Storage Utilization on Range Query Performance

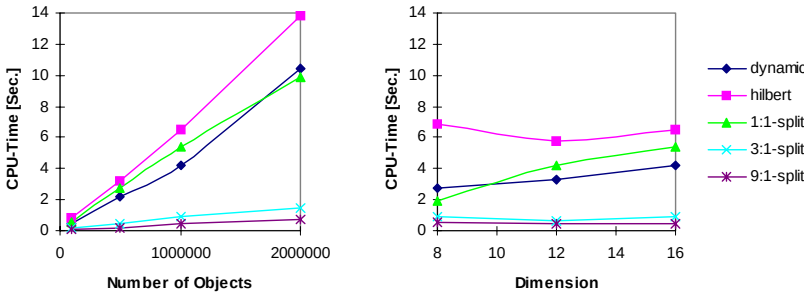


Figure 90: CPU-Time for Executing Range Queries

0.03%. The left diagram shows 16-dimensional indexes with between 100,000 and two million objects while the right diagram shows dimensions varying from 8 to 16 with a constant number of objects (1,000,000). The highest improvement (16.8) could be measured in the highest dimension and the largest database.

Next, we evaluated the influence of the storage utilization. Figure 89 shows the page accesses for a database with 1,000,000 points in a 16-dimensional data space over varying storage utilizations. Dynamic construction leads to a storage utilization of about 65%, whereas our implementation of Hilbert construction was limited to the 100%-factor. In contrast, using our technique, we are able to control the storage utilization and we varied it from 60% to 80%. As expected, it turned out that a higher storage utilization is beneficial. The performance, however, is only improved by low factors up to 30%. Therefore, the storage utilization is obviously not responsible for our good improvement factors presented so far.

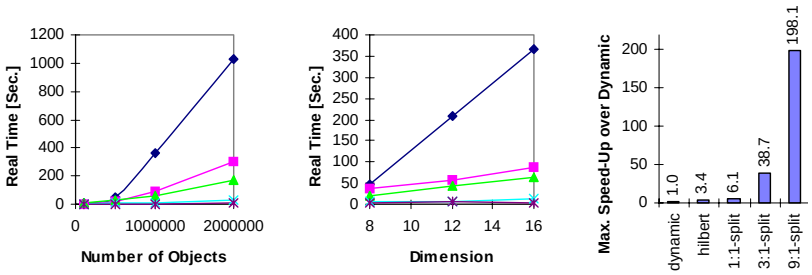


Figure 91: Real Time for Executing Range Queries

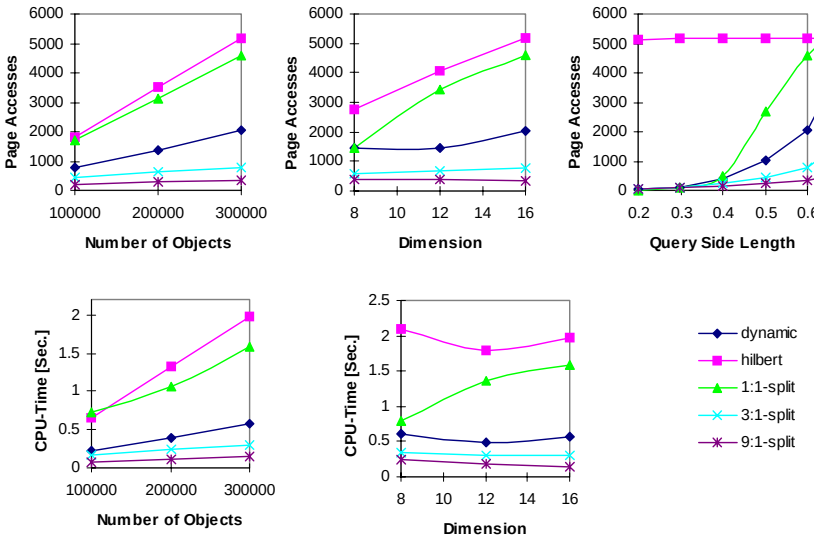


Figure 92: Experiments on Real Data (Text Descriptors)

Our next aim is to evaluate the performance impact in terms of CPU-time. Figure 90 again shows on the left side fixed dimension (16) and on the right side fixed database size (1,000,000 objects) with a selectivity of about 0.03%. The diagrams are widely congruent with the diagrams of figure 88 showing the page accesses. Therefore, we can assume that the absorbed CPU power is directly proportional with the number of page accesses. The improvement factor for CPU power reaches a value of 13.6.

Nevertheless, range query evaluation is clearly disk I/O bound as can be seen in figure 91. Here we measured the real time for query execution, encompassing CPU-time and the times for disk I/O which are predominant. It is remarkable that, in contrast to the experiments counting page accesses, the balanced splitting bottom-up method outperforms the dynamic construction, too, and that the improvement factors are one order of magnitude higher than the improvement factors for the page accesses. This is due to the much better disc clustering of our construction method. Data pages in a common subtree of the index are laid out contiguously on disk. These pages have often to be loaded commonly, such that disk head movements are often avoided. In contrast, if a dynamic index structure splits a page, one of the resulting new pages occupies the place of the old

page whereas the second page is appended at the end of the file. Thus, neighboring pages are rather declustered than clustered.

In a last series of experiments, we determined the behavior of our technique on real data, stemming from an information retrieval application. We used 300,000 feature vectors in a 16-dimensional data space which were converted from substring descriptors. The results confirm our previous results on synthetic data and are presented in figure 92. Unfortunately, the number of objects in our database was not high enough to yield similarly impressive improvement factors as with two million synthetic points. The improvement factors grow again with increasing dimension and increasing database size and reach a factor of 5.8.

Chapter 7
Optimized Declustering for Parallel Query Processing

In the preceding chapters, we proposed various techniques for the performance improvement of high-dimensional indexes. These techniques accelerate the range search and the nearest neighbor search in the case of a moderate dimensionality of the data space by large factors. Moreover, these techniques open the facility of efficiently indexing data spaces which cannot be managed by conventional indexing structures due to the high dimension.

Both our experiments and our analytical considerations provided in chapter 3 imply, however, that there exists a dimension for which index structures do not yield satisfactory performance, even if our improvement techniques from the preceding chapters are applied. To overcome this problem, we propose to exploit parallelism.

7.1 Introduction

Experiments with specialized high-dimensional index structures such as the TV-tree [LJF 95] and the X-tree [BKK 96] show significant performance improvements for point queries, but unfortunately only limited performance improvements for nearest-neighbor queries (cf. figure 93).

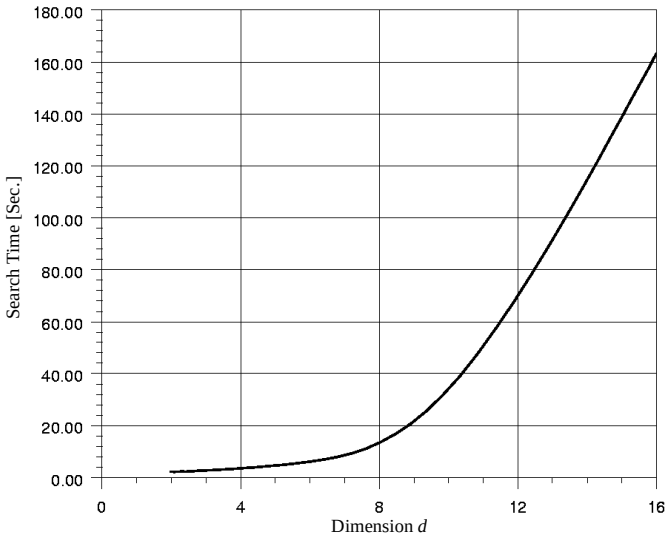


Figure 93: Nearest-Neighbor Queries in High Dimensions (X-tree)

In this chapter, we propose a new parallel method for fast nearest-neighbor search in high-dimensional feature spaces. This technique was published in a preliminary version [BBB+ 97]. In section 7.2, we first review the relevant literature. The core problem of designing a fast parallel nearest-neighbor algorithm is to find an adequate declustering algorithm which distributes the data onto the disks such that the data which has to be read when executing a query are distributed as equally as possible among the disks. Unfortunately, the known declustering methods such as the Disk Modulo [DS 82], FX [KP 88], and Hilbert Declustering [FB 93] have been designed to support different query types (range queries and partial match queries). Therefore, those techniques do not allow an optimal declustering for nearest-neighbor queries in high-dimensional spaces. In contrast, our new declustering method has been optimized based on the special properties of parallel nearest-neighbor search in high-dimensional spaces (cf. section 7.2.1) and therefore provides a near-optimal distribution of the data items among the disks (cf. section 7.2.2). The basic idea of our data declustering technique is to assign the buckets which correspond to different quadrants of the data space to different disks. We show that this problem is equivalent to a special case of the graph coloring problem (cf. section

7.3.1). Then, we develop a simple but efficient algorithm which solves the special case of the graph coloring problem and shows that our algorithm - in contrast to other declustering methods - guarantees that all buckets corresponding to neighboring quadrants are assigned to different disks (cf. section 7.3.2). A surprising result is that the number of disks necessary for the near-optimal declustering is a linearly bound staircase function which is optimal up to rounding (cf. section 7.3.2). We provide extensions of our algorithm considering an arbitrary number of disks and highly clustered data distributions (cf. section 7.3.3). Finally, in section 7.4, we evaluate our method using large amounts of uniformly distributed and real data (up to 40 MBytes) with varying dimension, and compare it with the best known data declustering method, the Hilbert curve. Our experiments show that our method provides a near-linear speed-up and a constant scale-up, and it outperforms the Hilbert approach by a factor of up to 5.

7.2 Parallel Nearest-Neighbor Search

The core problem of parallel nearest-neighbor search is the distribution of data among the available disks which is usually called the *declustering problem*. In the following, we denote the number of disks by n and the i-th disk by d_i.

The simplest method for distributing data is *round robin* where each disk d_i gets the data items $\{v_j | j \bmod n = i\}$. Figure 94 shows the speed-up of a parallel nearest-neighbor search (referred to as NN in all subsequent figures) and a parallel search for 10 nearest neighbors (10-NN) using the round robin data distribution on 1 MByte of uni-

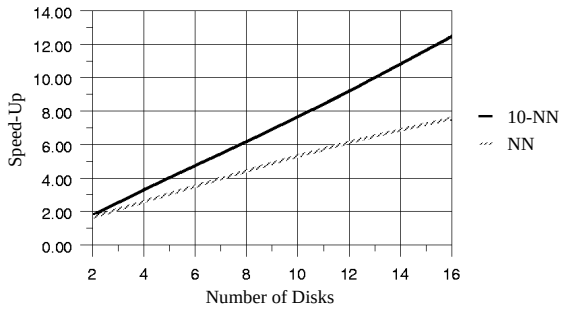


Figure 94: Speed-Up of Parallel Nearest-Neighbor Search (Round Robin)

formly distributed 15-dimensional data and uniformly distributed query points. In our experiment, the speed-up increases nearly linear with the number of disks. This simple experiment shows that nearest-neighbor search can be improved considerably by using parallelism.

More complex algorithms solving the declustering problem have been proposed in the literature. Using an equi-distant grid, all these algorithms divide the data space into equi-sized buckets *b* which may be characterized by the position of the bucket in the d-dimensional grid ($c_0, c_1, ..., c_{d-1}$). A bucket characterized by $b[c_0, c_1, ..., c_{d-1}]$ describes a partition of the data space having the shape of a hyperrectangle and containing a certain number of data objects. A declustering algorithm *DA* can be described as a mapping from the bucket characterization to a disk number.

A rather simple declustering algorithm is the disk modulo method of Du and Sobolewski [DS 82]. The *disk modulo* method uses the mapping

$$DM(c_0, c_1, \ldots, c_{d-1}) = \left(\sum_{l=0}^{d-1} c_l \right) \bmod n.$$

Kim and Pramanik [KP 88] improved the disk modulo method and presented the *FX* distribution method which has been specifically designed to support partial match queries. Kim and Pramanik distribute the buckets using a bitwise XOR operation. Slightly simplified, the FX method can be defined as the mapping

$$FX(c_0, c_1, \ldots, c_{d-1}) = \operatorname{XOR}_{l=0}^{d-1} c_l \bmod n.$$

In [FB 93], Faloutsos and Bhagwat apply the Hilbert curve to the declustering problem. The Hilbert curve maps a *d*-dimensional space to a 1-dimensional space. For mapping a point in the data space to a disk, the Hilbert value of the point is determined and the data point is stored on the disk corresponding to the Hilbert value. More formally, the i-th disk gets the bucket

$$HI(c_0, c_1, \ldots, c_{d-1}) = Hilbert(c_0, c_1, \ldots, c_{d-1}) \bmod n.$$

Since the Hilbert curve preserves spatial neighborhood as far as possible, the mapping provides a good declustering. Faloutsos and Bhagwat compared their method to various methods such as the disk modulo and the FX technique. The experimental results reported in [FB 93] show that the Hilbert approach clearly outperforms the other methods for

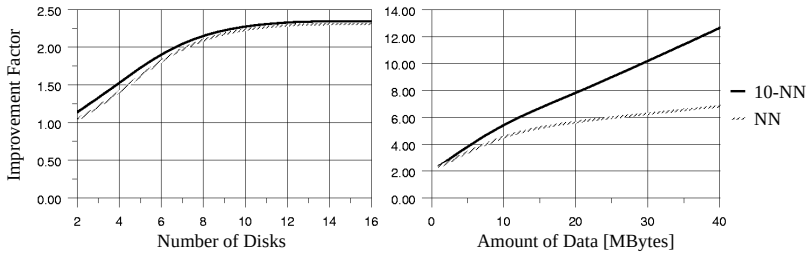


Figure 95: Improvement of Hilbert over Round Robin

range queries in two-dimensional spaces. However, to our knowledge, none of the methods has been designed or tested for high-dimensional feature spaces and for nearest-neighbor queries. Therefore, in our first experiments we used the most promising technique, the Hilbert curve. The experiments show that the Hilbert approach provides a much better declustering for nearest-neighbor queries in high-dimensional spaces than the round robin method. Figure 95 depicts the improvement of the Hilbert approach over the round robin declustering. Note that the improvement increases, both with an increasing number of disks, and with an increasing amount of data. In section 7.2.2, however, we show that all the methods described in this section including the Hilbert method do not provide an adequate data distribution for nearest-neighbor queries in high-dimensional spaces.

7.2.1 Effects in High-Dimensional Spaces

To find a good declustering algorithm, we have to consider several special effects occurring in high-dimensional spaces and their consequences for nearest-neighbor queries. In this section, we therefore analyze nearest-neighbor query processing in high-dimensional space and derive the requirements for an optimal declustering. For the following considerations, we assume uniformly distributed data and uniformly distributed query points.

During nearest-neighbor search, any NN-algorithm has to examine all data pages intersecting the so-called *NN-sphere* (cf. figure 96). The NN-sphere is a *d*-dimensional hypersphere having the query point as the centre and a radius equal to the distance from the query point to the nearest-neighbor. Unfortunately, according to [BBKK 97], the radius of the NN-sphere increases rapidly with increasing dimension of the data space,

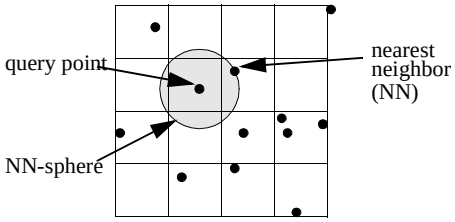


Figure 96: NN-Sphere

and therefore, the number of partitions any sequential algorithm has to access also increases rapidly. The increase of the radius depends on the bucket size, the number of data items and the dimension. However, the dimension is the most important parameter.

Declustering algorithms such as the disk modulo method or the FX method assume a partitioning of the data space into buckets. In the 2-dimensional case, the data space is partitioned many times in each direction, for example to obtain 10,000 buckets, the space is divided 100 times in x-direction and 100 times in y-direction. If we consider a 16-dimensional space, a complete *binary* partitioning of the space would already produce 65,536 partitions. Thus, in high-dimensional spaces it is not possible to consider more than a binary partitioning. In addition, the usage of a finer partitioning would produce many underfilled buckets. For the following considerations, we therefore assume each dimension of the space to be split exactly once. Thus, from our point of view, the buckets are the quadrants of the data space. The bucket coordinates $(c_0, c_1, ..., c_{d\text{-}1})$ can then be seen as binary values and $(c_0, ..., c_{d\text{-}1})$ may be represented as a bit-string. Note that $(c_0, c_1, ..., c_{d\text{-}1})$ with $c_i \in \{0, 1\}$ corresponds to the binary representation of the corresponding grid partition stored in the bucket. We use this property to define an unambiguous bucket number *bn* which will be the basis for our algorithm presented in section 7.3.2.

Definition 13: Bucket Number

Given a bucket b characterized by $(c_0, c_1, ..., c_{d\text{-}1})$ with $c_i \in \{0, 1\}$, $0 \le i < d$. The bucket number *bn* is defined as

$$bn(b) = \sum_{i=0}^{d-1} c_i \cdot 2^i .$$

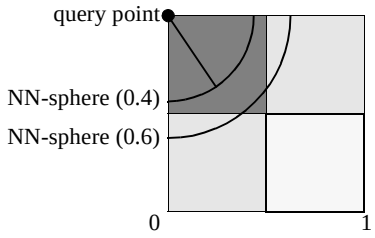


Figure 97: Partitions Affected by the Search when Increasing the NN-sphere

7.2.2 Declustering for Nearest-Neighbor Search

The goal of each declustering algorithm is to distribute the buckets which are involved in an *arbitrary* search to different disks. For the parallel nearest-neighbor search, this means that the partitions intersecting the NN-sphere should be distributed to different disks. If all disks are equally involved in the search, the speed-up is maximal.

Figure 97 illustrates the effects of an increasing NN-sphere using a two-dimensional example. Let us assume that the query point is located in the upper left corner of the data space. If the radius of the NN-sphere is less than 0.5, only the bucket containing the query point has to be accessed (the upper left bucket in figure 97). Thus, only the disk which stores the bucket is involved in the search process and any declustering technique provides the same result. If the radius of the NN-sphere is 0.6, however, two other buckets are involved in the search (the lower left and the upper right bucket in figure 97). Obviously, for obtaining a good speed-up, the three buckets involved in the search should be distributed to different disks. Note that in high-dimensional space, this observation holds for most queries even if the query point is not located exactly in a corner of the data space but on a lower-dimensional surface, e.g. a two-dimensional surface (cf. figure 5).

Generalizing this result to the *d*-dimensional case, a good declustering technique must assure that adjacent buckets are assigned to different disks. From the example in figure 97, we can derive that not only directly adjacent buckets (such as the upper left and upper right bucket) have to be considered, but also indirectly neighboring buckets (such as the lower left and the upper right bucket). This can be formalized as follows**:**

Definition 14: Direct and Indirect Neighbors

Given two buckets b and c.

b and c are *direct neighbors*, $b \sim_d c$, if and only if

$$\exists i: \begin{cases} b_k \neq c_k, & \text{iff } k = i \\ b_k = c_k, & \text{otherwise} \end{cases}, \text{ where } (0 \leq i, k \leq (d-1)).$$

b and c are *indirect neighbors*, $b \sim_i c$, if and only if

$$\exists (i,j), i \neq j: \begin{cases} b_k \neq c_k, & \text{iff } k = i \text{ or } k = j \\ b_k = c_k, & \text{otherwise} \end{cases}, \text{ where } (0 \leq i, j, k \leq (d-1)).$$

Intuitively, two buckets b and c are direct neighbors if their coordinates differ in one dimension, and the remaining (d-1) coordinates are identical. Note that this definition of neighborhood implies that applying the binary exclusive-or-function (XOR) to direct neighboring buckets b and c results in a bit-string of the form 0^*10^*. Analogously, applying the XOR function to indirectly neighboring buckets results in a bit-string of the form $0^*10^*10^*$. Note further that considering more than one level of indirection would produce a huge amount of neighboring buckets. An algorithm considering i levels of indirection in d-dimensional space would have to assure that

$$1 + \sum_{k=1}^{i} \binom{d}{k}$$

buckets are equally distributed over the disks. For two levels of indirection in a 16-dimensional space, for example, the number of buckets would be

$$1 + \sum_{k=1}^{2} \binom{16}{k} = 1 + 16 + 120 = 137.$$

Therefore, we restricted our definition of neighboring buckets to direct and indirect neighbors. Another important observation is the following: Direct neighbors share a common 1-dimensional surface of the data space, whereas indirect neighbors share a 2-dimensional surface.

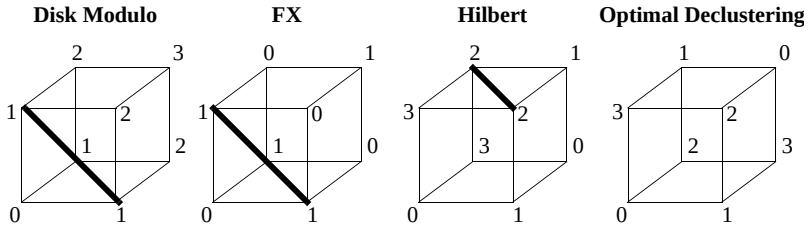


Figure 98: Disk Modulo, FX and Hilbert are not Near-Optimal Declustering Techniques

Using the above definitions, we can define a *near-optimal* declustering as a declustering which guarantees that all direct and indirect neighboring buckets are assigned to different disks. We use the term *near-optimal* because an optimal declustering technique would have to guarantee that arbitrary queries are handled by different disks. This however would require to consider arbitrary neighbors – not only direct and indirect neighbors.

Definition 15: Near-Optimal Declustering

A declustering algorithm DA is near-optimal if and only if for any two buckets *b* and *c* and for any dimension *d* of the data space:

$$b \sim_d c \rightarrow DA(b) \neq DA(c) \text{ and } \quad b \sim_i c \rightarrow DA(b) \neq DA(c).$$

As we show in our experimental evaluation, our definition of a near-optimal declustering algorithm is close to the optimum, i.e. it provides a high speed-up and a nearly constant scale-up. The following lemma shows that the known declustering techniques do not provide a near-optimal declustering.

Lemma 12: Sub-Optimality of Disk Modulo, FX and Hilbert Declustering

The disk modulo, the FX, and the Hilbert declustering techniques are not near-optimal declustering algorithms.

Proof (Lemma 12)

The validity of lemma 12 can be shown by a simple three-dimensional counter-example (cf. figure 98). The numbers in the corner of each cube denote the disk number the corresponding bucket is assigned to. The thick line in each cube shows indirect neighbors which are assigned to the same disk. The right most portion of figure 98 demon-

strates the existence of a near-optimal declustering. Note that there exist more than one colliding pair of indirect neighbors, which however are not shown in figure 98.

❑

7.3 Near-optimal Declustering for Nearest-Neighbor Queries

In this section, we present a new declustering technique which is near-optimal according to definition 15. The basic idea of our technique is to transform the declustering problem into an equivalent graph-coloring problem so that buckets correspond to vertices, neighborhood-relations to edges, and disks to colors. We propose a simple but efficient algorithm for solving the graph-coloring problem. To show that our declustering technique is near-optimal, we prove that our graph-based algorithm assigns different colors to connected vertices in the graph. The number of colors (disks) required by our algorithm is a linearly bounded staircase function which is optimal up to rounding. Furthermore, we describe some extensions of our method, allowing the method to be used in a wide range of real applications, i.e. on data with various data distributions and dimensionalities, and an arbitrary number of disks.

7.3.1 Declustering as a Graph Coloring Problem

In order to transform the declustering problem into a graph coloring problem, we first define the disk assignment graph. The disk assignment graph is an undirected graph in which buckets correspond to vertices. Neighborhood relationships between buckets correspond to edges.

Definition 16: Disk Assignment Graph

The disk assignment graph $G_d = (V, E)$ for a d-dimensional data space is an undirected graph where $V=\{0, ..., 2^d-1\}$ is the set of bucket numbers and $E = \{(b, c) \mid b, c \in V$ and $b \sim_d c$ or $b \sim_i c\}$ is the set of direct and indirect neighborhood relationships.

Since our definition of the edges includes both direct and indirect neighbors, it is obvious that an algorithm which assigns different colors to connected vertices provides a near-optimal declustering. Thus, we reduce the declustering problem to an equivalent graph coloring problem.

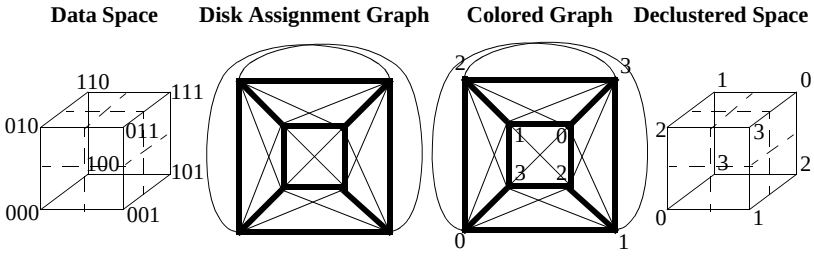


Figure 99: Disk Assignment Graph

Figure 99 shows the disk assignment graph G_3 for a three-dimensional data space. In the left partition of the figure, the data space with the corresponding buckets is depicted. In the middle of the figure, the corresponding disk assignment graph is shown with thick lines denoting direct neighbors and thin lines denoting indirect neighbors. The disk assignment graph G_3 may be colored using 4 colors. Transforming the graph back, we get a near-optimal declustering of the space (cf. right part of figure 99). Obviously, a lower bound of d+1 colors is required to color a graph G_d because each vertex has d directly neighboring vertices and at least all directly neighboring vertices must have pairwise different colors. It is a well-known fact from graph theory [Big 89] that the graph coloring problem for arbitrary graphs (including the determination of the required number of colors) is a hard problem which has not been solved in polynomial time yet and therefore, it is believed that the problem is NP-complete. Nevertheless, we are able to exploit some regularities in our graph to develop a simple but efficient coloring algorithm.

7.3.2 The Vertex Coloring Algorithm

In this section, we introduce an algorithm to determine the vertex color (i.e. the disk number) for a given vertex (i.e. the bucket number). After describing the algorithm, we prove that our algorithm assigns different colors to connected vertices and we provide a formula for the number of colors required by our algorithm.

The basic idea of our algorithm is to determine for a vertex b all positions in its binary representation which are equal to 1. Incrementing these positions by 1, each position can again be interpreted as a binary number. These numbers are combined by the XOR function. Interpreting the resulting binary number as a decimal number, we finally ob-

```
int col (int c) {
        int i ;
        int c = 0 ;
        for (i = 0 ; i < dimension ; i++)
                if (bit_set (i, c) )
                        c = c ^ (i+1);       // XOR
        return c ;
}
```

Figure 100: Vertex Coloring Algorithm

tain the corresponding vertex color. We will motivate later why we have to increment the positions before combining them using XOR. Intuitively, the reason is that otherwise the information about dimension '0' would not be considered by the vertex coloring function.

For example, let us assume a given vertex $c = 5 = 101_2$ in a disk assignment graph G_3 (representing a 3-dimensional data space). As the bits c_0 and c_2 are set, the positions to be considered are 0 and 2. Incrementing the positions by one, we obtain (2+1)=3 and (0+1)=1. We combine the binary representations 011_2 (=3) and 001_2 (=1) by the XOR function and obtain 011_2 XOR $001_2 = 010_2$. Interpreting this binary number as a decimal number, we get $010_2 = 2_{10}$. The color of vertex 5 is therefore 2. Figure 100 shows the vertex coloring algorithm in an algorithmic pseudocode. It is obvious from the algorithm that the color of an arbitrary vertex may be determined in $O(d)$ time. The following formal definition provides a very compact form of the algorithm.

Definition 17: Vertex Coloring Function

Given a vertex number c in binary representation $c_{d-1}, ..., c_0$. The corresponding vertex color is

$$\text{col}(c) = \left(\operatorname*{XOR}_{i=0}^{d-1} \left\{ \begin{array}{ll} i+1 & \text{if } c_i = 1 \\ 0 & \text{otherwise} \end{array} \right) \right)_{10}.$$

In the following, we show that our vertex coloring function *col* guarantees that vertices which are connected in the disk assignment graph are colored differently. Our proof is divided into three lemmata. First, we prove the distributivity of *col* and XOR. Then, we prove that vertices which are connected by an edge representing direct neighborhood are

colored differently, and finally we prove the same for edges representing indirect neighborhood.

Lemma 13: Distributivity of *col* and XOR

$$\forall b \ \forall c: \quad col(b) \text{ XOR } col(c) = col(b \text{ XOR } c)$$

Proof (Lemma 13)

$$col\,(b) \text{ XOR } col\,(c) =$$

$$= \underset{i=0}{\overset{d-1}{\text{XOR}}} \left(\begin{cases} i+1 & \text{if } b_i = 1 \\ 0 & \text{otherwise} \end{cases} \right) \text{ XOR } \underset{i=0}{\overset{d-1}{\text{XOR}}} \left(\begin{cases} i+1 & \text{if } c_i = 1 \\ 0 & \text{otherwise} \end{cases} \right)$$

$$= \underset{i=0}{\overset{d-1}{\text{XOR}}} \left(\begin{cases} 0 \text{ XOR } 0 & \text{if } b_i = 0 \text{ and } c_i = 0 \\ i+1 \text{ XOR } 0 & \text{if } b_i = 1 \text{ and } c_i = 0 \\ 0 \text{ XOR } i+1 & \text{if } b_i = 0 \text{ and } c_i = 1 \\ i+1 \text{ XOR } i+1 & \text{if } b_i = 1 \text{ and } c_i = 1 \end{cases} \right)$$

$$= \underset{i=0}{\overset{d-1}{\text{XOR}}} \left(\begin{cases} i+1 & \text{if } b_i \text{ XOR } c_i = 1 \\ 0 & \text{otherwise} \end{cases} \right)$$

$$= col\,(b \text{ XOR } c).$$

❑

Using lemma 13, we now prove that vertices which are connected by an edge representing direct neighborhood are colored differently. We make use of some algebraic laws which are valid for the XOR function, especially the associativity, commutativity and the following equivalences:

$$a \text{ XOR } b = 0 \Leftrightarrow a = b$$
$$a \text{ XOR } b = \text{a} \Leftrightarrow b = 0$$

Lemma 14: Coloring of Direct Neighbors

Two vertices *b* and *c* which are connected by an edge representing a direct neighborhood are colored differently.

Proof (Lemma 14)

As the vertices b and c are differing in exactly one bit, say bit j, b XOR c is of the form 0^*10^* with only bit j set (cf. definition 14). Therefore, using the definition of the vertex coloring function, we may derive that $col(b \text{ XOR } c) = j + 1 \neq 0$. Thus,

$$\begin{aligned} col(c) &= \\ &= col(b \text{ XOR } b \text{ XOR } c) && \text{(since } b \text{ XOR } b = 0 \text{ and } 0 \text{ XOR } c = c) \\ &= col(b) \text{ XOR } col(b \text{ XOR } c) && \text{(according to lemma 13)} \\ &= col(b) \text{ XOR } (j + 1) && \text{(since only bit } j \text{ is set in } b \text{ XOR } c) \\ &\neq col(b) && \text{(since otherwise, } (j+1) \text{ must be 0)} \end{aligned}$$

❑

Lemma 15: Coloring of Indirect Neighbors

Two vertices b and c which are connected by an edge representing an indirect neighborhood are colored differently.

Proof (Lemma 15)

According to definition 14, b XOR c has the form $0^*10^*10^*$ with a bit set at the positions i and j, $i \neq j$ and $col(b \text{ XOR } c) = (i+1) \text{ XOR } (j+1)$ which cannot be zero since $i+1 \neq j+1$. Thus,

$$\begin{aligned} col(c) &= col(b) \text{ XOR } col(b \text{ XOR } c) \\ &= col(b) \text{ XOR } (i + 1) \text{ XOR } (j + 1) \\ &\neq col(b) \end{aligned}$$

❑

Lemma 16: Near-Optimal Declustering by *col*-Function

The vertex coloring function *col* for the declustering of a d-dimensional data space is near-optimal.

Proof (Lemma 16)

According to definition 15, a declustering algorithm DA is near-optimal if and only if

$$b \sim_d c \rightarrow DA(b) \neq DA(c)$$

and

$$b \sim_i c \rightarrow DA(b) \neq DA(c).$$

We proved that our algorithm *col* assigns different colors to connected vertices in the disk assignment graph. As vertices are connected if the corresponding buckets are direct or indirect neighbors, the function *col* guarantees that neighboring buckets are assigned to different disks.

❑

So far, we have shown that our algorithm computing the vertex color assigns pairwise different colors to all neighbors of any given vertex and therefore provides a near-optimal declustering.

Now, we want to determine how many colors are necessary for a d-dimensional data space. It seems to be obvious that any vertex coloring algorithm solving the disk assignment problem must use at least d+1 colors since each vertex and its d direct neighbors have to be colored differently. This means that no algorithm exists which is better than linear in the number of dimensions. We show in our next lemma that the number of colors provided by our algorithm is a linearly bounded staircase function which is optimal up to rounding.

Lemma 17: Number of Colors Required by the Color Assignment Function

The number of colors required by the color assignment function is $||\lceil d+1 \rceil||$, where $||\lceil ... \rceil||$ denotes the rounding to the next-higher power of two, formally

$$||\lceil a \rceil|| = 2^{\lceil \log_2 a \rceil}.$$

Proof (Lemma 17)

First, we prove that our algorithm never generates a vertex corresponding to a color greater or equal to:

$$2^{\lceil \log_2(d+1) \rceil}.$$

According to *col*, the color of a vertex is a XOR-combination of some numbers from the set {1, ..., d} (cf. definition 17). The binary representation of d has exactly $\lceil \log_2(d+1) \rceil$ bits. Therefore, the XOR-combination cannot create a number with more bits, and the highest possible number with $\lceil \log_2(d+1) \rceil$ bits is

$$2^{\lceil \log_2(d+1) \rceil} - 1.$$

Next, we prove that all color numbers in the interval

$$[0, 2^{\lceil \log_2(d+1) \rceil} - 1]$$

are generated by the color assignment function. According to *col*, the vertex of the origin (0, 0, ..., 0) has color number zero (*col*(0) = 0). For any other vertex color *c*, bounded by the interval above, an appropriate bucket number *b* can easily be constructed such that *col*(*b*)=*c* by the following algorithm: If bit *j* is set in *c*, then set also bit 2^j-1 in *b* and reset all other bits in *b*. The result is a valid bucket number for the *d*-dimensional hypercube as can be seen from the following argumentation: We know that

$$j < \lceil \log_2(d+1) \rceil .$$

Therefore, *b* has less than $2^{\lceil \log_2(d+1) \rceil - 1}$ bits,

and thus

$$b < 2^{2^{\lceil \log_2 d+1 \rceil - 1}} = 2^{\frac{1}{2} \cdot |\lceil d+1 \rceil|}.$$

As *b* has to be smaller than 2^d in order to be a legal vertex number for a *d*-dimensional hypercube,

$$2^{\frac{1}{2} \cdot |\lceil d+1 \rceil|} \leq 2^d \Leftrightarrow |\lceil d+1 \rceil| \leq 2d .$$

This is guaranteed since a power of two is always between a number and its double:

$$\forall d \in |\mathrm{N} : \exists k \in |\mathrm{N}_0 : d < 2^k \leq 2d .$$

$|\lceil d+1 \rceil|$ cannot be rounded up to anything above 2*d*. If the bits with the numbers $2^{j_i} - 1$ for some j_i are set in *b*, then according to definition 17, the color number *col*(*b*) is

$$col(b) = \operatorname*{XOR}_i (2^{j_i}),$$

which combines to *c*. Altogether, we have proven that our algorithm uses exactly the colors with the numbers

$$0 \leq c < |\lceil d+1 \rceil| .$$

❑

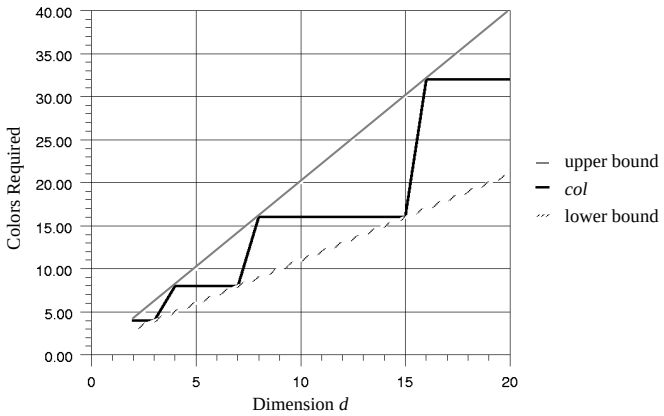


Figure 101: Number of Colors Required by *col*

The number of colors required to solve the vertex coloring problem is a staircase-function (cf. figure 101) above the line (d + 1) which has already been identified to be a lower bound for the number of colors. For lower dimensions, we have verified by enumerating all possible color assignments that there is no method which uses fewer colors than our staircase function. We conjecture that this is also true for higher dimensions. In any case, we are able to give the linear upper and lower bounds for the staircase function. As already mentioned, the lower bound is d+1. The upper bound is 2d, as may be seen with the same argument already used in lemma 17: There is always a number corresponding to power of two between a number d and its double 2d. Therefore, $|\lceil d + 1 \rceil|$ cannot be higher than 2d for $d \in |\mathrm{N}$.

7.3.3 Extensions of our Declustering Technique

In this section, we propose two extensions of our declustering technique. First, we describe an adaptation of our method for supporting an arbitrary number of disks and second, we describe an extension of our method for highly clustered data.

An important requirement for any parallel approach is to support an arbitrary number of processing units (disks). For our problem, this means that we have to adapt our algorithm to work with an arbitrary number of disks since our vertex coloring function *col* requires the optimal number of 2^i disks. We now describe a simple method for reducing

the number of disks required; in a first step by a factor of 2 (preserving that direct neighbors are assigned to different disks), and in a second step to an arbitrary number.

As we can easily derive from the 3-dimensional example in figure 99, there exists no near-optimal declustering algorithm using less than 4 disks for the 3-dimensional case. As a consequence, reducing the number of colors generated by our function *col* may induce that indirectly neighboring buckets are assigned to the same disk. Our extension of the function *col*, however, guarantees that most directly neighboring buckets are still assigned to different disks. The extension reduces the number of required disks by a factor of 2. The basic idea of our extension is to map one half of the colors to their binary-complementary color. For example, to decluster an 8-dimensional data space, the function *col* requires $C = 16$ disks numbered from 0 to 15. In our first reduction step, we map the colors 8..15 to the colors 0..7 such that 8 is mapped to 7, 9 is mapped to 6, ..., and 15 is mapped to 0. Obviously, our extended algorithm requires a total number of $(C/2)$ disks. Note that this mapping guarantees that most directly neighboring buckets are still assigned to different disks. Intuitively, we map the colors to their complement because complementary colors have the maximal Hamming distance, i.e. differ in a maximum number of bits.

In the general case, let us assume that we have n disks available where $n < C$. If $n \leq C/2$, we map each color c which is larger than $C/2$ to its binary complement. Thus, we have only $C/2$ colors left. Note that the most significant bit of these $C/2$ colors is the bit 0. If n is smaller than $C/4$, we again map the colors greater than $C/4$ to their complements, while, however, ignoring the most significant bit. This process is repeated until $n \leq C/2^k$. The number of colors required by the algorithm is now $C/2^{k-1}$. In order to obtain exactly n colors, we again map the highest $C/2^{k-1} - n$ colors to their complements. Recording the mappings in a table, we are able to determine the disk number from the color number *col* by a single table look-up.

Another extension of our declustering techniques focuses on highly clustered data. In real applications, high-dimensional data is usually not distributed uniformly. If the data points are highly clustered, i.e. most data points are located in one quadrant of the hypercube, our technique as described so far would assign most data points to a single disk. Although in most applications such an extreme case will not occur, we have to consider data distributions where many points are assigned to a few disks, i.e. the amount of data stored on the disks differs largely.

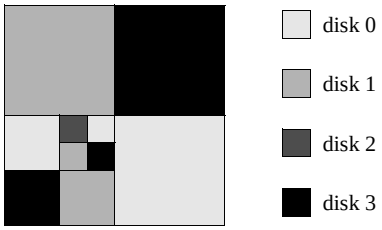


Figure 102: Recursive Declustering

A first solution to this problem is to use a statistical measure, the α-quantile, to divide the buckets. Instead of splitting each dimension in the middle, we determine the 0.5-quantile of each dimension and use the values as split values for determining the bucket boundaries. One may argue that we do not know the data distribution a priori and are therefore not able to determine the correct 0.5-quantile in advance. To solve the problem, we dynamically adapt the 0.5-quantile by recording the distribution according to the previous 0.5-quantile, i.e. counting the number of data points below and above the split value. If the ratio of these two numbers extends a certain threshold, we reorganize our data distribution using the new 0.5-quantile for each dimension.

If the data points are highly correlated, the usage of a one-dimensional quantile is not sufficient. This situation is detected if the one-dimensional α-quantile does not change but the disks are loaded unbalanced, nonetheless. Our strategy for this case is to recursively decluster the overloaded buckets of the data space. The optimal declustering means to decluster all overloaded buckets. This, however, would require an amount of $O(2^d)$ of storage space which cannot be handled for higher dimensions. Therefore, our approach recursively declusters all buckets of a single disk in one step using our *col* declustering function (cf. figure 102), which means a transfer of the affected data to another disk. Note that we may have to apply the recursive declustering more than once if necessary. As first experiments show, permuting the colors using a simple heuristic when going to the next level of recursion provides good speed-ups (cf. figure 108).

Note that our parallel nearest-neighbor search is completely dynamical. This means that we are able to support insertions, updates, and deletions without any a priori knowledge of the data. However, for highly clustered or correlated data, a reorganization may be necessary.

7.4 Experimental Results

In order to show the efficiency and practical relevance of our declustering technique, we performed an extensive experimental evaluation of our technique and compared it to the Hilbert declustering which is the most promising declustering method designed for low-dimensional data spaces. All experiments have been computed on a workstation cluster of 16 HP710 workstations, each having 32 MBytes of main memory and several hundred MBytes of secondary storage. All programs have been implemented in C++ as templates to support different types of data objects. In order to analyze our method, we integrated our declustering technique and the Hilbert declustering into a parallel version of the X-tree [BKK 96].

In our experiments, we used three types of data: Fourier points corresponding to contours of industrial parts (d=8..15), text data corresponding to substrings of a large set of texts (d=15), and uniformly distributed points (d=8..15). The total amount of data used in our experiments was about 800 MBytes. The block size used is 4 KBytes. In order to measure the performance of our technique, we determined the disk which accesses most pages during query processing. We used the search time of this disk as the search time of the whole parallel X-tree. Each experiment has been performed 10 times and the average of the 10 experiments is used as the reported search time. In order to compute the speed-up, we compared the search time of the parallel X-tree with a sequen-

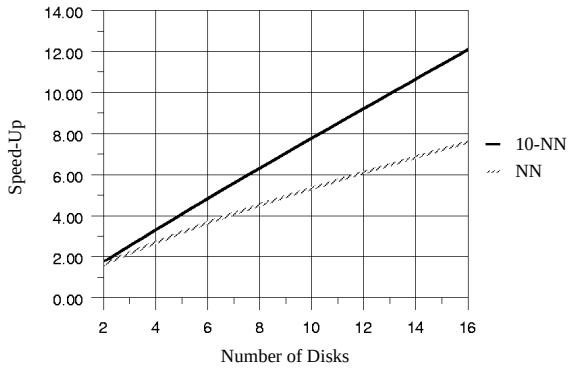


Figure 103: Speed-Up of Our Technique on Uniformly Distributed Data (1 MByte)

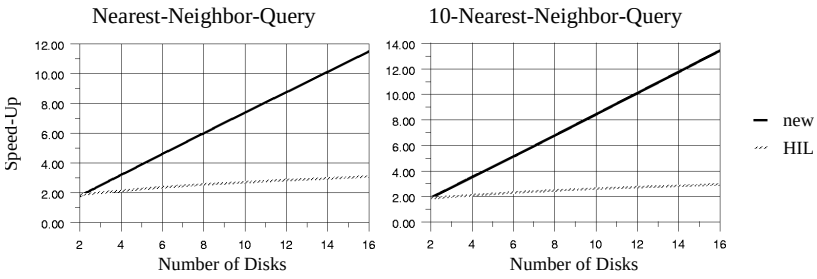


Figure 104: Speed-Up of Our Technique and Hilbert Declustering (Fourier Points)

tial X-tree using the original implementation of [BKK 96]. In the following figures, “new” denotes our technique, whereas “HIL” denotes the Hilbert approach.

Our first objective was to show the linear speed-up of our new method. We performed an experiment on 1 MByte of uniformly distributed data (d=15) with varying numbers of disks (cf. figure 103). In performing a nearest-neighbor query, the speed-up reaches a value of 8 for 16 disks for a nearest-neighbor query. For 10-nearest-neighbor queries, the speed-up increases up to a value of 12 for 16 disks. In both experiments, the speed-up was nearly linear.

Since one cannot assume a uniform data distribution for real life applications, we used real data for our further experiments. Again, we investigated the speed-up of our technique and compared it to the Hilbert declustering for a nearest-neighbor query and a 10-nearest-neighbor query. Figure 104 shows the speed-up of our technique and the Hilbert curve on 40 MBytes of 15-dimensional Fourier points. Obviously, both techniques achieve a near-linear speed-up for both query types. However, our technique clearly outperforms the Hilbert curve which reaches only 19% of the optimal speed-up using 16 disks. Figure 105 shows the improvement of our technique over the Hilbert approach in the same experiment. The factor linearly increases with the number of disks and approaches a value of 5 for 16 disks. Note that this is due to the fact that the Hilbert curve does not provide a near-optimal declustering.

Next, we made experiments to measure the scale-up of our technique, i.e. we increased the number of disks and increased the total amount of data proportionally. In particular, we increased the number of disks from 2 to 16 while increasing the amount of

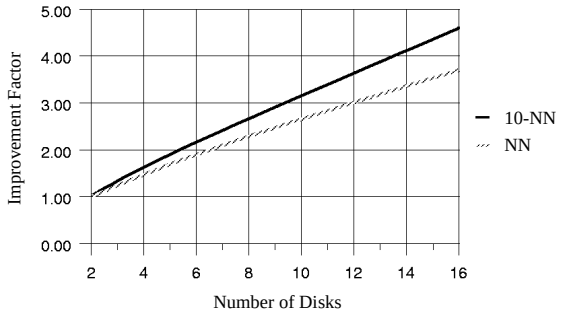


Figure 105: Improvement Factor over Hilbert Declustering (Fourier Points)

data from 1 to 8 MBytes. Figure 106 depicts the result of this experiment. The total search time is nearly constant for both, nearest-neighbor queries and 10-nearest-neighbor queries. The experiment shows that our technique scales well when increasing the problem size.

In addition to the Fourier data, we also used text descriptors for our experiments. The text descriptors are feature vectors characterizing substrings of large sets of various documents given in ASCII format. Again, we compared our technique to the Hilbert approach. Figure 107 shows a total search time of 771 ms for our technique in contrast to 1683 ms for the Hilbert approach for a nearest-neighbor query (improvement of 2.18) on 1MByte of 15-dimensional text descriptors. For the 10-nearest-neighbor query the improvement of our technique increased to 2.99.

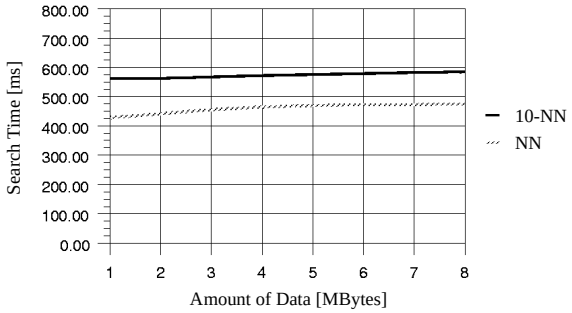


Figure 106: Scale-Up on NN Queries and 10-NN Queries (Fourier Points)

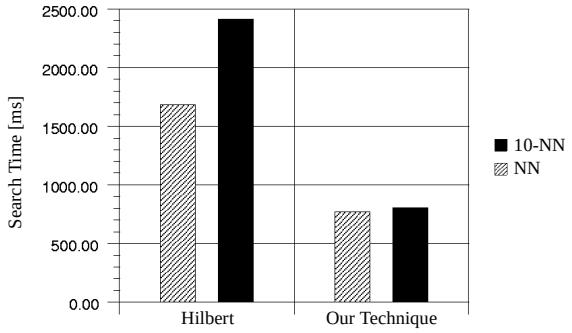


Figure 107: Total search time of our technique and the Hilbert curve (Text Data)

In section 7.3.3, we proposed several extensions of our technique. The first extension, the adaptation to an arbitrary number of disks, has been used for all experiments presented in this chapter which use a varying number of disks. The second extension of our technique has also been implemented and tested. Figure 108 depicts the results of these experiments. The experiments have been performed with 40 MBytes of 15-dimensional Fourier points. The Fourier points represent a set of variants of CAD-parts and are highly clustered. The original technique yielded a total search time of 537.6 ms for a nearest-neighbor query, whereas the extension reduced the total search time to 137.7 ms. The large improvement factor of 3.9 is due to the fact that a large amount of data items is located in the same quadrant of the data space and therefore assigned to a single disk. Note that only one recursive declustering step was necessary in the experiments.

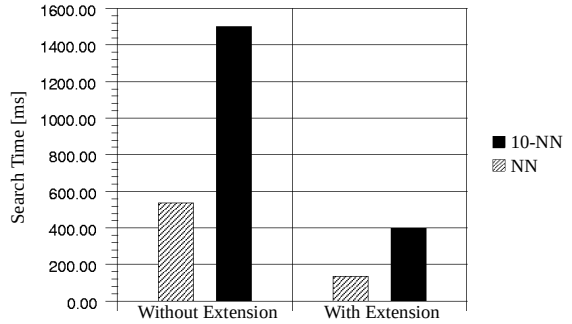


Figure 108: Effect of Recursive Declustering

Chapter 8
Indexing Ultra-High-Dimensional Feature Spaces

In the preceding chapters, we proposed various techniques to improve index structures for high-dimensional query processing. It turned out, however, that sufficient performance is only achieved in cases of a moderate dimensionality. A result of recent research activities [BBKK 97, BKK 96, WJ 96] is that basically none of the querying and indexing techniques also performs sufficiently well on data spaces which are of a very high dimension such as 100. The only approach taken to solve this problem for larger queries was parallelization (cf. chapter 7). In this chapter, however, we will tackle the problems leading to the so-called curse of dimensionality.

8.1 Introduction

A variety of new index structures [KS 97, LJF 95], cost models [BBKK 97, FBF 77] and query processing techniques [BEK+ 98, BBK+ 98] have been proposed. Most of the index structures are extensions of multidimensional index structures adapted to the requirements of high-dimensional indexing. Thus, all these index structures are restricted with respect to the data space partitioning. Additionally, they suffer from the well-known drawbacks of multidimensional index structures such as high costs for insert and delete operations and a poor support of concurrency control and recovery.

Motivated by these disadvantages of state-of-the-art index structures for high-dimensional data spaces, we developed the Pyramid-Technique. The Pyramid-Technique is based on a special partitioning strategy which is optimized for high-dimensional data. The basic idea is to divide the data space such that the resulting partitions are shaped like peels of an onion. Such partitions cannot efficiently be stored by R-tree-like index structures. We can achieve such partitions, however, by dividing the d-dimensional space into $2d$ pyramids having the center point of the space as their top. In a second step, the single pyramids are cut into slices parallel to the basis of the pyramid to form the data pages. As we will show, both analytically and experimentally, this strategy outperforms other partitioning strategies when processing range queries. Furthermore, we will analytically show that range query processing using our method is not affected by the so-called "curse of dimensionality", i.e. the performance of the Pyramid-Technique does not deteriorate when going to higher dimensions. Instead, the performance improves for increasing dimension. Note that this analytical result is obtained for hypercube shaped queries and uniform data distribution. Queries which touch the boundary of the data space or very skewed queries are handled less efficiently. However, as we will show in the experimental section of this paper, even slightly skewed queries can be handled efficiently.

Another advantage of the Pyramid-Technique is the fact that we use a mapping from the given d-dimensional data space to a 1-dimensional space in order to achieve the mentioned onion-like partitioning. Therefore, we can use a B^+-tree [BM 77, Com 79] to store the data items and take advantage of all the nice properties of B^+-trees such as fast insert, update and delete operations, good concurrency control and recovery, easy implementation and re-usage of existing B^+-tree implementations. The Pyramid-Technique can easily be implemented on top of an existing DBMS.

The rest of this chapter is organized as follows: In section 8.2, we introduce the Pyramid-Technique and show how the index construction is performed. In section 8.3 we describe query processing using the Pyramid-Technique in detail. Then, we analyze in section 8.4 the benefits of the Pyramid-Technique. To improve the performance of the Pyramid-Technique in case of real data, we propose some extensions of the Pyramid-Technique in section 8.5. Finally, we present a variety of experiments demonstrating the practical impact of our technique. A discussion of the weaknesses and limitations of the Pyramid-Technique will conclude the chapter. The material presented in this paper has partly been previously published [BBK 98b].

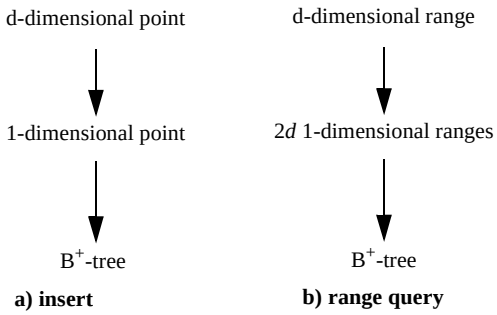


Figure 109: Operations on Indexes

8.2 The Pyramid-Technique

The basic idea of the Pyramid-Technique is to transform the *d*-dimensional data points into 1-dimensional values. For storing and accessing the values, we use an efficient index structure such as the B^+-tree [BM 77, Com 79]. Potentially, any order-preserving one-dimensional access method can be used. Operations such as insert, update, delete or search operations are performed by using the B^+-tree. Figure 109 depicts the general procedure of an insert operation and the processing of a range query. In both cases, the *d*-dimensional input is transformed into some 1-dimensional information which can be processed by the B^+-tree. Note that although we index our data using a 1-dimensional key, we store *d*-dimensional points *plus* the corresponding 1-dimensional key in the leaf nodes of the B^+-tree. Therefore, we do not have to provide an inverse transformation. The transformation itself is based on a specific partitioning of the data space into a set of *d*-dimensional pyramids. Thus, in order to define the transformation, we first explain the data space partitioning of the Pyramid-Technique.

8.2.1 Motivation

The basic motivation of space partitioning using the Pyramid-Technique is related to the technique of unbalanced splitting presented in chapter 6. The index architecture, however, is totally different from our previous proposal. Index construction, maintenance and

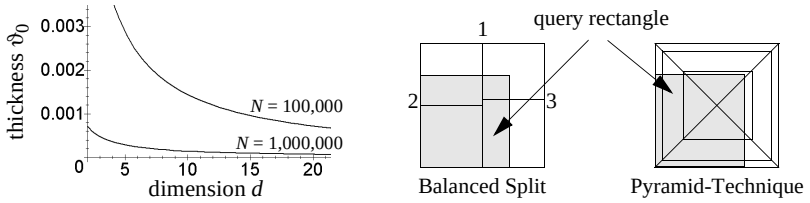


Figure 110: Partitioning Strategies

query processing are also completely new since the Pyramid-Technique transforms the d-dimensional points into a one-dimensional embedding rather than organizing a rectilinear directory. Figure 110 on the right shows the partitions of an index using the pyramid technique. The page regions are shaped like peels of an onion. Under uniformity assumption, it is very likely that pages near by the boundary of the data space are very thin. The outermost peel, for example, contains $2d$ data pages and, thus, $C_{\text{eff}} \cdot 2d$ data points. The thickness ϑ_0 of the peel is determined such that it contains an expectation of $C_{\text{eff}} \cdot 2d$ data points, resulting in the equation:

$$V_{\text{DS}} - (\sqrt[d]{V_{\text{DS}}} - \vartheta_0)^d = \frac{C_{\text{eff}} \cdot 2d}{N}.$$

If we assume the data space to be normalized to the unit hypercube ($V_{\text{DS}} = 1$), we get the following result for the thickness of the outermost peel:

$$\vartheta_0 = 1 - \sqrt[d]{1 - \frac{C_{\text{eff}} \cdot 2d}{N}}.$$

Figure 110 depicts in the diagram on the left side ϑ_0 for varying dimension. The thickness decreases with increasing dimension. The middle and right side of figure 110 compare balanced splitting with the Pyramid-Technique. For the outermost peel, it is for a reasonable selectivity impossible that the query touches both partitions on opposite sides. As the outermost peel is very thin (typically in the order of $10^{-3}..10^{-4}$ of the side length of the data space, it is even unlikely that any of the partitions on the outermost peel is intersected by a query. The next peel under the outermost peel is affected with a slightly higher probability. But the total expectation of page accesses is still very low. We will analyze this effect in depth in section 8.4.

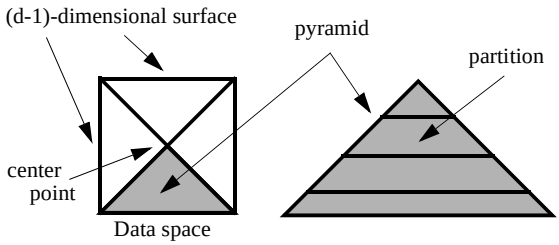


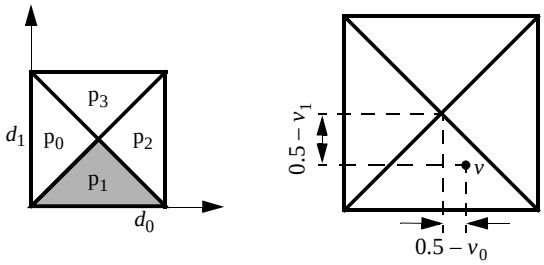
Figure 111: Partitioning the Data Space into Pyramids

8.2.2 Data Space Partitioning

The Pyramid-Technique partitions the data space in two steps: in the first step, we split the data space into 2*d* pyramids having the center point of the data space (0.5, 0.5, ..., 0.5) as their top and a (*d*-1)-dimensional surface of the data space as their base. In a second step, each of the 2*d* pyramids is divided into several partitions, each corresponding to one data page of the B^+-tree. In the 2-dimensional example depicted in figure 111, the space has been divided into 4 triangles (the 2-dimensional analogue of the *d*-dimensional pyramids) which all have the center point of the data space as top and one edge of the data space as base (figure 111 left). In a second step, these 4 partitions are split again into several data pages parallel to the base line (figure 111 right). Given a *d*-dimensional space instead of the 2-dimensional space, the base of the pyramid is not a 1-dimensional line such as in the example, but a (*d*-1)-dimensional hyperplane. As a cube of dimension *d* has 2*d* (*d*-1)-dimensional hyperplanes as a surface, we obviously obtain 2*d* pyramids.

Numbering the pyramids as in the 2-dimensional example in figure 112a, we can make the following observations which are the basis of the partitioning strategy of the Pyramid-Technique: All points located on the *i*-th (*d*-1)-dimensional surface of the cube (the base of the pyramid) have the common property that either their *i*-th coordinate is 0 or their $(i - d)$-th coordinate is 1. We observe that the base of the pyramid is a (*d* - 1)-dimensional hyperplane, because one coordinate is fixed and (*d* - 1) coordinates are variable. On the other hand, all points *v* located in the *i*-th pyramid p_i have the common property that the distance in the *i*-th coordinate from the center point is either smaller than the distance of all other coordinates if $i < d$, or larger if $i \geq d$. More formally:

$$\forall j,\ 0 \leq j < d, j \neq i\text{: } (|0.5 - v_i| \leq |0.5 - v_j|) \qquad if (i < d)$$
$$\forall j,\ 0 \leq j < d, j \neq (i - d)\text{: } (|0.5 - v_{(i-d)}| \geq |0.5 - v_j|)\ \ if (i \geq d)$$



a) numbering of pyramids **b**) point in pyramid

Figure 112: Properties of Pyramids

Figure 112b depicts this property in two dimensions: all points located in the lower pyramid are obviously closer to the center point in their d_0-direction than in their d_1-direction. This common property provides a very simple way to determine the pyramid p_i which includes a given point v: we only have to determine the dimension i having the maximum deviation $|0.5 - v_i|$ from the center. More formally:

Definition 18: Pyramid of a point *v*

A d-dimensional point v is defined to be located in pyramid p_i,

$$i = \begin{cases} j_{max} & if(v_{j_{max}} < 0.5) \\ (j_{max} + d) & if(v_{j_{max}} \geq 0.5) \end{cases}$$

$$\text{with } j_{max} = (j \,|\, (\forall k,\ 0 \leq (j,k) < d, j \neq k\text{: } |0.5 - v_j| \geq |0.5 - v_k|)) .$$

Note that all further considerations are based on this definition. Therefore, it is crucial for our technique.

Another important property is the location of a point v within its pyramid. This location can be determined by a single value which is the distance from the point to the center point according to dimension j_{max}. As this geometrically corresponds to the height of the point within the pyramid, we call this location height of v (cf. figure 113)

Definition 19: Height of a point *v*

Given a *d*-dimensional point *v*. Let p_i be the pyramid in which *v* is located according to definition 18. Then, the height h_v of the point *v* is defined as

$$h_v = \left|0.5 - v_{i \text{ MOD } d}\right| .$$

Using definition 18 and 19, we are able to transform a *d*-dimensional point *v* into a value $(i+h_v)$ where *i* is the index of the according pyramid p_i and h_v is the height of *v* within p_i. More formally:

Definition 20: Pyramid value of a point *v*

Given a *d*-dimensional point *v*. Let p_i be the pyramid in which *v* is located according to definition 18 and h_v be the height of *v* according to definition 19. Then, the pyramid value pv_v of *v* is defined as

$$pv_v = (i + h_v) .$$

Note that *i* is an integer and h_v is a real number in the range [0, 0.5]. Therefore, each pyramid p_i covers an interval of $[i, (i+0.5)]$ pyramid values and the sets of pyramid values covered by any two pyramids p_i and p_j are disjunct. Note further that this transformation is not injective, i.e. two points *v* and *v'* may have the same pyramid value. As mentioned above, we do not require an inverse transformation and therefore we do not require a bijective transformation.

8.2.3 Index Creation

Given the transformation determining the pyramid value of a point *q*, it is a simple task to build an index based on the Pyramid-Technique. In order to dynamically insert a point *v*, we first determine the pyramid value pv_v of the point and insert the point into a B^+-tree using pv_v as a key. Finally, we store the *d*-dimensional point *v* and pv_v in the according data page of the B^+-tree. Update and delete operations can be done analogously. Note

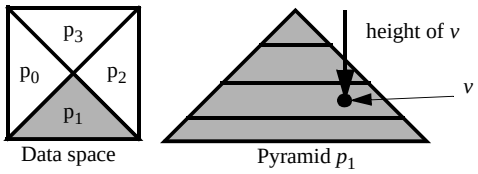


Figure 113: Height of a Point within its Pyramid

that B^+-trees can be bulk-loaded very efficiently, for example, when building a B^+-tree from a large set of data items. The bulk-loading techniques for B^+-trees can be applied to the Pyramid-Technique as well.

In general, the resulting data pages of the B^+-tree contain a set of points which all belong to the same pyramid and have the common property that their pyramid value lies in an interval given by the minimal and maximal key value of the data pages. Thus, the geometric correspondence of a single B^+-tree data page is a partition of a pyramid as shown in figure 113 (right).

8.3 Query Processing

In contrast to the insert, delete and update operation, query processing using the Pyramid-Technique is a complex operation. Let us focus on point queries first which are defined as *"Given a query point q, decide whether q is in the database"*. Using the Pyramid-Technique, we can solve the problem by first computing the pyramid value pv_q of q and querying the B^+-tree using pv_q. As a result, we obtain a set of d-dimensional points sharing pv_q as a pyramid value. Thus, we scan the set and determine whether the set contains q and output the result.

In case of range queries, the problem is defined as follows: *"Given a d-dimensional interval*

$$[q_{0_{min}}, q_{0_{max}}], ..., [q_{d-1_{min}}, q_{d-1_{max}}],$$

determine the points in the database which are inside the range". Note that the geometric correspondence of a multidimensional interval is a hyper-rectangle. Analogously to point queries, we face the problem to transform the d-dimensional query into a 1-dimensional query on the B^+-tree. However, as the simple 2-dimensional example depicted in figure 114 (left) demonstrates, a query rectangle may intersect several pyramids and the computation of the area of intersection is not trivial. As we also take from the example, we first have to examine which pyramids are affected by the query, and second, we have to determine the ranges inside the pyramids. The test whether a point is inside the ranges is based on a single attribute criterion (h_v between two values). Therefore, determining all such objects is a one-dimensional indexing problem. Objects outside the ranges are guaranteed not to be contained in the query rectangle. Points lying inside the ranges, are candidates for a further investigation. It can be seen in figure 114 that some of the candi-

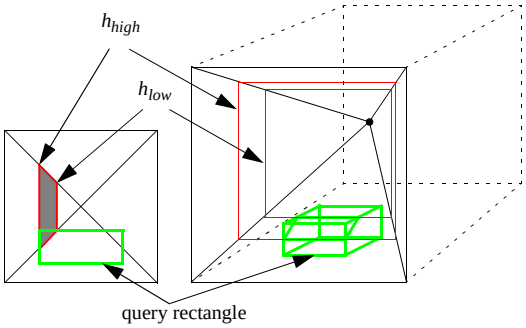


Figure 114: Transformation of Range Queries

dates are hits, others are false hits. Then, a simple point-in-rectangle-test is performed in the refinement step.

For simplification, we focus the description of the algorithm only on pyramids p_i where $i < d$, however, our algorithm can be extended to all pyramids in a straight-forward way. As a first step of our algorithm, we transform the query rectangle q into an equivalent rectangle $\hat{q}$ such that the interval is defined relative to the center point.

$$\hat{q}_{j_{min}} = q_{j_{min}} - 0.5 \text{ and } \hat{q}_{j_{max}} = q_{j_{max}} - 0.5, \forall j, 0 \le j < d.$$

Additionally, we interpret any point v mentioned in this section to be defined relative to the center point of the data space. Based on definition 18, we are able to determine if a pyramid p_i is affected by a given query $\hat{q}$. As we will see, we have to determine the absolute minimum and maximum of an interval which is defined as follows: Let $MIN(r)$ be defined as the minimum of the absolute values of an interval *r:*

$$MIN(r) = \begin{cases} 0 & \text{if } r_{min} \le 0 \le r_{max} \\ min(|r_{min}|, |r_{max}|) & \text{otherwise} \end{cases}.$$

Note that $|r_{min}|$ may be larger than $|r_{max}|$. Analogously, we define:

$$MAX(r) = max(|r_{min}|, |r_{max}|).$$

Lemma 18: (Intersection of a Pyramid and a Rectangle)

A pyramid p_i is intersected by a hyperrectangle $[\hat{q}_{0_{min}}, \hat{q}_{0_{max}}], \ldots, [\hat{q}_{d-1_{min}}, \hat{q}_{d-1_{max}}]$ if and only if

$$\forall j, 0 \leq j < d, j \neq i: \hat{q}_{i_{min}} \leq -MIN(\hat{q}_j).$$

Proof (Lemma 18)

The query rectangle intersects pyramid p_i if and only if there exists a point v inside the rectangle which falls into pyramid p_i. Thus, the coordinates $|v_j|$ of v must all be smaller than $|v_i|$. This, however, is only possible if the minimum absolute value in the query rectangle in dimension j is closer to the center point than $\hat{q}_{i_{min}}$ is to the center point. Lemma 18 follows from the fact that this must hold for all dimensions j.

❑

In the second step, we have to determine which pyramid values inside an affected pyramid p_i are affected by the query. Thus, we are looking for an interval $[h_{low}, h_{high}]$ in the range of [0, 0.5] such that the pyramid values of all points inside the intersection of the query rectangle and pyramid p_i are in the interval $[i+h_{low}, i+h_{high}]$. Figure 114 depicts this interval for two and three dimensions.

In order to determine h_{low} and h_{high}, we first restrict our query rectangle to pyramid p_i, i.e. we remove all points above the center point:

$$\widehat{q}_{i_{min}} = \hat{q}_{i_{min}}, \ \widehat{q}_{i_{max}} = min(\hat{q}_{i_{max}}, 0),$$

$$\widehat{q}_{j_{min}} = \hat{q}_{j_{min}}, \text{ and } \widehat{q}_{j_{max}} = \hat{q}_{j_{max}}, \text{ where } (0 \leq j < d), j \neq i.$$

Note that we restricted our considerations to the pyramids $p_0 .. p_{d-1}$. Therefore, the relevant values of $\hat{q}_{i_{min}}$ and $\hat{q}_{i_{max}}$ are negative. The effect of this restriction is depicted in a two-dimensional example in figure 115 (upper part).

The determination of the interval $[h_{low}, h_{high}]$ is very simple if the center point of the data space is included in the query rectangle, i.e. $\forall j, (0 \leq j < d): (\hat{q}_{j_{min}} \leq 0 \leq \hat{q}_{j_{max}})$. In this case, we simply use the extension of the query rectangle as a result, thus:

$$h_{low} = 0 \text{ and } h_{high} = MAX(\widehat{q}_i).$$

If the center point is not included in the query rectangle, we make the observation that $h_{high} = MAX(\widehat{q}_i)$, too. This is due to the fact that the query rectangle must contain at

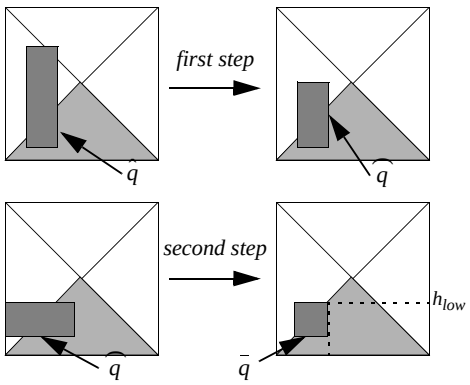


Figure 115: Restriction of Query Rectangle

least one point v such that $v_i = MAX(\widehat{q}_i)$ because otherwise there would be no intersection between the query rectangle and pyramid p_i.

In order to find the value h_{low}, we have to determine the minimum height of points inside the query rectangle and the pyramid p_i. As we consider points which are inside $\widehat{q}$ and inside p_i, we can intersect all intervals $[\widehat{q}_{j_{min}}, \widehat{q}_{j_{max}}]$ $(0 \le j < d), j \ne i$ with $[\widehat{q}_{i_{min}}, \widehat{q}_{i_{max}}]$ without affecting the value h_{low}. Then, the minimum of the *min*-values of all dimensions of the new rectangle $\bar{q}$ is equal to h_{low}. Figure 115 (lower part) shows an example of this operations. Obviously, the checkered rectangles on the left and the right side of each example are causing the same value h_{low}.

Lemma 19: (Interval of Intersection of Query and Pyramid)

Given a query interval $\hat{q}$ and an affected pyramid p_i, the intersection interval $[h_{low}, h_{high}]$ is defined as follows:

Case 1: $(\forall j, 0 \le j < d: (\hat{q}_{j_{min}} \le 0 \le \hat{q}_{j_{max}}))$

$$h_{low} = 0,$$

$$h_{high} = MAX(\widehat{q}_i).$$

Case 2: (otherwise)

$$h_{low} = min_{(0 \le j < d, j \ne i)}(\bar{q}_{j_{min}}) \ (*)$$

$$h_{high} = MAX(\widehat{q}_i)$$

$$\bar{q}_{j_{min}} = \begin{cases} max(MAX(\widehat{q}_i), MIN(\widehat{q}_j)) & \text{if } MAX(\widehat{q}_j) \geq MIN(\widehat{q}_i) \\ MIN(\widehat{q}_i) & \text{otherwise} \end{cases}$$

Proof (Lemma 19)

We will show for any point *v* which is located inside the query rectangle $\hat{q}$ and an affected pyramid p_i that the resulting query interval $[h_{high}, h_{low}]$ contains $|v_i|$. Note that we assumed *i* to be smaller than *d* and thus $v_i < 0$. Therefore, we have to show that

$$h_{low} \leq |v_i| \leq h_{high}.$$

1. $|v_i| \leq h_{high}$:

This holds because we choose h_{high} such that $|v_i| \leq MAX(\widehat{q}_i) = h_{high}$.

2. $h_{low} \leq v_i$:

If $\hat{q}$ contains the center point, we have $h_{low} = 0 \leq |v_i|$.

Otherwise, $|v_i| \geq |v_j|, (\forall j, (0 \leq j < d))$ because *v* is inside the pyramid *i*. On the other hand, $v_j \geq \hat{q}_{j_{min}}, \forall j, (0 \leq j < d)$ because *v* is inside the query rectangle and $v_j \geq \widehat{q}_{j_{min}}$ because all coordinates of *v* are negative for $0 \leq i < d$. Thus, $|v_j| \geq MIN(\widehat{q}_j), (\forall j, 0 \leq j < d)$.

Additionally, $|v_i| \geq MIN(\widehat{q}_i)$ because of the same reasons. Assembling the two results, we derive: $|v_i| \geq max(MIN(\widehat{q}_i), MIN(\widehat{q}_j)), \forall j, 0 \leq j < d$. From equation (*), however, follows that $\bar{q}_{j_{min}} \geq h_{low}$. Thus, we finally obtain that $|v_i| \geq max(MIN(\widehat{q}_i), MIN(\widehat{q}_j)) \geq h_{low}$

❑

Lemma 18 and 19 imply the simple query processing algorithm depicted in figure 116.

8.4 Theoretical Analysis

In contrast to our assumptions in chapter 3, we focus in this section on window queries which are completely contained in the data space and extend our model to reflect such a query distribution. For simplicity, we assume a uniform, independent distribution of data points in the *d*-dimensional unit hypercube and hypercube shaped window queries with

```
Point_Set PyrTree::range_query(range q)
{
  Point_Set res;

  for (i = 0; i < 2d; i++) {
    if (intersect(p[i], q) {
    // using lemma 18
      determine_range(p[i], q, h_low, h_high);
      // using lemma 19

      cs = btree_query(i+h_low, i+h_high);
      for (c = cs.first; cs.end; cs.next){
        if (inside(q, c))
          res.add(c);
      }
    }

  }
  return res;
}
```

Figure 116: Processing Range Queries (Algorithm)

side length q. The query anchor which is in this case the lower left corner of the query window by definition, is therefore taken randomly from the hypercube $[0 .. 1 - q]^d$ to make sure that the query window is completely contained in the data space.

8.4.1 Analysis of Balanced Splitting

We start with the analysis of balanced splitting for our query distribution. Our assumption is (like in chapter 3) that in high-dimensional query processing the data space cannot be split in each dimension, but only in a number d' of dimensions with

$$d' = \log_2(\frac{N}{C_{eff}}).$$

As the range queries are completely contained in the data space, there is no need to consider boundary effects for the queries. If s is the selectivity, i.e. the ratio of objects to be retrieved, then the side-length q simply corresponds to the d-th root of s:

$$q = \sqrt[d]{s}.$$

For a 20-dimensional range query with a selectivity of 0.01%, we obtain a side length of $q = 0.63$ which is larger than half of the extension of the data space in this dimension. For

a page b_i given by its lower and upper bounds $lb_{i,j}$ and $ub_{i,j}$ ($0 \le j < d$), we can extend our formula for the access probability

$$P_{\text{range}}(q) = \prod_{j=0}^{d-1} (ub_{i,j} - lb_{i,j} + q)$$

to take our new query distribution into account:

$$P_{\text{window}}(q) = \prod_{j=0}^{d-1} \frac{\min(ub_{i,j}, 1-q) - \max(lb_{i,j} - q, 0)}{1-q} .$$

The minimum and maximum are necessary to cut the parts of the Minkowski sum exceeding the data space, whereas the denominator $(1 - q)$ is due to fact that the stochastic "event space" of the query anchor is not [0, 1] but rather [0, $1 - q$]. The model for balanced splits can be simplified if the number of data pages is a power of two. Then, all pages have the extension 0.5 in d' dimensions, accommodated in the lower or the upper half of the data space, and full extension in the remaining dimensions. By C_{eff} we denote the effective (average) capacity of a data page. It is dependent on d. As in our special case, all pages have the same access probability and thus the expected value of data page accesses is:

$$E_{\text{window}}(d, q, N) = \frac{N}{C_{\text{eff}}} \cdot \min\left(1, \left(\frac{0.5}{1-q}\right)^{\log_2\left(\frac{N}{C_{\text{eff}}}\right)}\right).$$

Note that we require the minimum to assure that the expected value does not exceed the total number of data pages and that we are able to ignore the remaining ($d - d$') dimensions because the extension of the data pages in these dimensions is 1.

8.4.2 Analysis of the Pyramid Technique

Now we are going to determine a cost estimation for the Pyramid-Technique. We restrict ourselves to the cost for processing hypercube shaped range queries having a side length larger than 0.5 to achieve a reasonable selectivity for high-dimensional queries. In this case, the center of the data space is always contained in the query and therefore, our window query is transformed into a set of exactly $2d$ one-dimensional range queries with

$$h_{low} = 0 \text{ and } h_{high} = MAX(\widehat{q}_i).$$

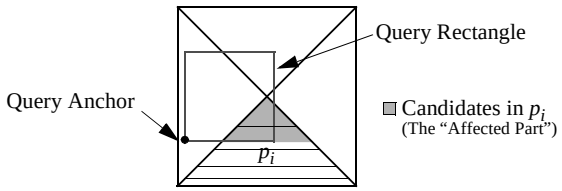


Figure 117: Modeling the Pyramid-Technique

We do not need the concept of the Minkowski sum here because we analyze the performance of one-dimensional interval queries. However, we have to take into account that, in contrast to the points of the database, the pyramid values are not uniformly distributed.

In the first step of our model, we determine an expected value for the amount of data in each pyramid which has to be accessed during query processing (the size of the candidate set). We consider the lower boundaries of the query rectangle $QA = (q_{0_{min}}, ..., q_{d-1_{min}})$ as the anchor point of the query. QA is obviously taken from the multidimensional interval $QAI = [0, 1-q]^d$ to guarantee that the whole query is located inside the data space. Therefore, the height h_{high} in the pyramid p_i is uniformly distributed in the interval $H_i = [q-0.5, 0.5]$ (cf. figure 117). We call the part of the hyper-pyramid, starting with $h_{low} = 0$ and ending with h_{high} (underlaid in grey in figure 117), the affected part of the pyramid. The volume of the affected part can be determined by using the fact that it is the $2d$-th part of a hypercube with side length $2 \cdot h_{high}$:

$$V(h_{high}) = \frac{(2 \cdot h_{high})^d}{2 \cdot d}.$$

From this volume of the affected part for a given h_{high}, we can also determine the expected value by forming an average over all possible positions of h_{high} in the interval H_i. Thus, we have to integrate over h_{high} and then divide the result by the size of the interval H_i which yields the following integral formula:

$$E_V(d, q) = \frac{\int_{q-0.5}^{0.5} V(h_{high}) \; d\, h_{high}}{0.5 - (q-0.5)}.$$

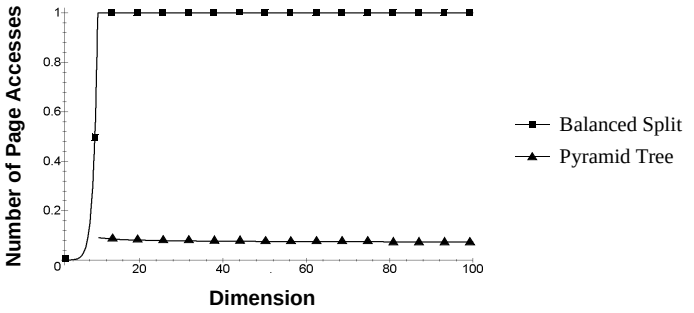


Figure 118: Range Queries Using the Pyramid Technique and Balanced Splitting

The integral can be evaluated and simplified to:

$$E_V(d, q) = \frac{1 - (2q - 1)^{d+1}}{4d \cdot (1 - q) \cdot (d + 1)}.$$

As $E_V(d, q)$ is the expected volume of the affected part for a query of the size q in a single pyramid under the uniformity assumption that

$$2d \cdot E_V(d, q) \cdot (N/2d) = E_V(d, q) \cdot N$$

is the expected total number of objects in the affected parts of all pyramids.

These objects are the candidates for an exact-geometry test of d-dimensional range containment (cf. figure 117). Since it is unlikely that the affected part is perfectly aligned with a break between two subsequent pages, the question is, how many data pages are occupied by the candidates. Note that all candidates belong to a single interval of pyramid values and therefore, the candidates are stored contiguously on the data pages. Thus, assuming a pagination with the effective page capacity C_{eff}, we have to descend the directory of the B^+-tree for each pyramid to find the object with the lowest pyramid value in each pyramid. This object may be located anywhere inside a data page. Then, we have to read a run with the length of $E_V(d, q) \cdot N$ objects which occupies $E_V(d, q) \cdot N/C_{\text{eff}}$ data pages. The last object is, again located somewhere on a data page with an equal probability of every position on the page. On average, we have to read half a page before and after the run, respectively. Therefore, the required number of accesses to data pages for all $2d$ pyramids is:

$$E_{\text{pyramidtree}}(d, q, N) = \frac{2d + N \cdot (1 - (2q - 1)^{d+1})}{2C_{\text{eff}}(d) \cdot (d + 1) \cdot (1 - q)}.$$

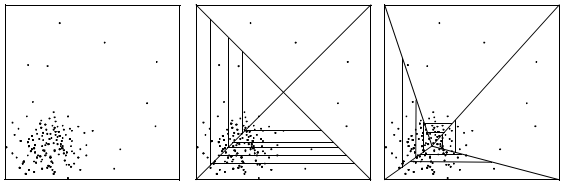


Figure 119: Effect of Clustered Data

The number of accesses to directory pages is $2d$ times the height of the B^+-tree $\log_{C_{\text{eff,dirpg}}}(N / C_{\text{eff}})$ and can be neglected because the directory fits into the cache. We made the same assumption in the model for balanced splitting.

8.4.3 Comparison

Figure 118 depicts the performance of the Pyramid-Technique as predicted by our model and, in comparison, the estimated cost when using balanced splitting. The Pyramid-Technique does not reveal any performance degeneration in high dimensions. Note that we achieved this result by assuming hypercube shaped queries which are uniformly distributed over the data space and, therefore, the result only holds for this query type.

8.5 The Extended Pyramid-Technique

All our considerations presented so far were based on the assumption that the data is uniformly distributed. However, data produced by real-life applications does not behave this way. Therefore, the question arises, how to adapt the Pyramid-Technique to real data. Let us consider the following scenario: What happens to the Pyramid-Technique if most of the data is located in one corner of the data space (figure 119 left). Obviously, only a few pyramids (in the extreme case only one) will contain most of the data while the other pyramids are nearly empty. This, however, will result in the suboptimal space partitioning depicted in the example in figure 119 (middle). Obviously, partitioning is suboptimal because we can assume real-life queries to be similarly distributed as the data itself. Under this realistic assumption, a much better partitioning for the same data set is shown in figure 119 (right).

The basic idea of the extended Pyramid-Technique is to achieve such a partitioning by transforming the data space such that the data cluster is located in the center point (0.5, ..., 0.5) of space. Thus, we have to map the given data space to the canonical data space $[0, 1]^d$ such that the *d*-dimensional median of the data is mapped to the center point. Note that we only have to assure that the median of the data roughly coincides with the center point of the data space. The presence of clusters distributed over the space does not cause a problem for our technique. However, we only apply the transformation to determine the pyramid values of points and query rectangles, but not to the points itself. Therefore, we do not have to apply the inverse transformation to our answer set.

As the computation of the *d*-dimensional median is a hard problem, we use the following heuristic to determine an approximation of the *d*-dimensional median: We maintain a histogram for each dimension to keep track of the median in this dimension. The *d*-dimensional median is then approximated by the combination of the *d* one-dimensional medians. Obviously, the approximated *d*-dimensional median may be located outside the convex hull of the data cluster. As our experiments showed, this effect occurs very rarely and therefore the performance of our algorithms is not affected. The computation of the median can either be done dynamically in case of dynamic insertions, or once in case of a bulk-load of the index.

Given the *d*-dimensional median *mp* of the data set, we define a set of *d* functions t_i, $0 \le i < (d-1)$ transforming the given data space in dimension *i* such that the following conditions hold:

1. $t_i(0) = 0$
2. $t_i(1) = 1$
3. $t_i(mp_i) = 0.5$
4. t_i: $[0, 1] \rightarrow [0, 1]$

The three conditions are necessary to assure that the transformed data space still has an extension of $[0..1]^d$ (1. and 2.), and that the median of the data becomes the center point of the data space (3.). Condition 4. assures that each point in the original data space is mapped to a point inside the canonical data space. The resulting functions t_i can be chosen as an exponential function such that:

$$t_i(x) = x^r.$$

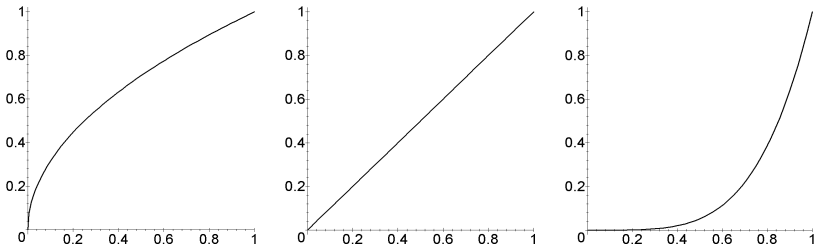


Figure 120: Transformation Functions t_i

Obviously, conditions 1., 2., and 4. are satisfied by x^r, $r \geq 0$, $0 \leq x \leq 1$. In order to determine the parameter r, we have to satisfy condition 3.: $t_i(mp_i) = 0.5 = mp_i^r$. Thus,

$$r = -\frac{1}{\log_2(mp_i)} \text{ and } t_i(x) = x^{-\frac{1}{\log_2(mp_i)}}.$$

Now, in order to insert a point v into an index using the extended Pyramid-Technique, we simply transform v into a point $v'_i=t_i(v_i)$ and determine the pyramid value $pv_{v'}$. Then, we insert v using $pv_{v'}$ as a key value as described in section 8.2.3. In order to process a query, we first transform the query rectangle q (or query point) into a query rectangle q' such that $q'_{i_{min}}=t_i(q_{i_{min}})$ and $q'_{i_{max}}=t_i(q_{i_{max}})$. Note that q' is a rectangle because we applied independent transformations to each dimension. Next, we use the algorithm presented in section 8.3, to determine the intervals of affected pyramid values and query the B^+-tree. As a result, we obtain a set of non-transformed d-dimensional points v which we test with the original query rectangle q. Note that we used the transformations t_i only to determine the pyramid value but we have not transformed the points itself.

If we dynamically build an index, the situation may occur that the first 10% of inserted points have a median different from that of the other 90% of the data. More general, we have to handle the situation that the median changes during the insertion process. To handle this case, we maintain the current median by maintaining a histogram for each dimension and rebuild the index if the distance of the current median to the center point exceeds a certain threshold. Note that rebuilding the index is not too expensive because we use a bulk-load technique for B^+-trees. In order to determine a good threshold, we use

the value $th = (\sqrt{d})/4$ because the maximum distance from any point to the center point is $(\sqrt{d})/2$ and, therefore, the adapting process is guaranteed to terminate after a logarithmic number of steps. Note further that the probability that the median shifts and therefore the index has to be reorganized decreases with an increasing percentage of inserted data items. Therefore, a reorganization occurs very rarely in practice. Furthermore, our experiments showed that a slightly shifted median has a negligible influence on the performance of the Pyramid-Technique.

8.6 Experimental Evaluation

To demonstrate the practical impact of the Pyramid-Technique and to verify our theoretical results, we performed an extensive experimental evaluation of the Pyramid-Technique and compared it to the following competitive techniques:

- X-tree [BKK 96]
- Hilbert-R-tree [KF 94]
- Sequential Scan [WSB 98].

The Hilbert-R-tree has been chosen for comparison, because the Hilbert-curve and other space filling curves can be used in conjunction with a B-tree in a so-called one-dimensional embedding. Since the Pyramid-Technique also incorporates a very sophisticated one-dimensional embedding, the Hilbert R-tree appeared to us as a natural competitive method.

Recently, the criticism arose that index-based query processing is generally inefficient in high-dimensional data spaces [BGRS 98], and that sequential scan processing yields a better performance in this case. Therefore, we included the sequential scan in our experiments. We will confirm the observation that the sequential scan outperforms the X-Tree and the Hilbert R-Tree for high dimensionalities, but we will also see that our new technique outperforms the sequential scan in all experiments performed.

For clarity, we state our assumption that all relevant information is stored in the various indexes, as well as in the file used for the sequential scan. Therefore, no additional accesses to fetch objects for presentation or further processing are needed in any of the techniques applied in our experiments.

Our experiments have been computed on HP 9000/780 workstations with several GigaBytes of secondary storage.

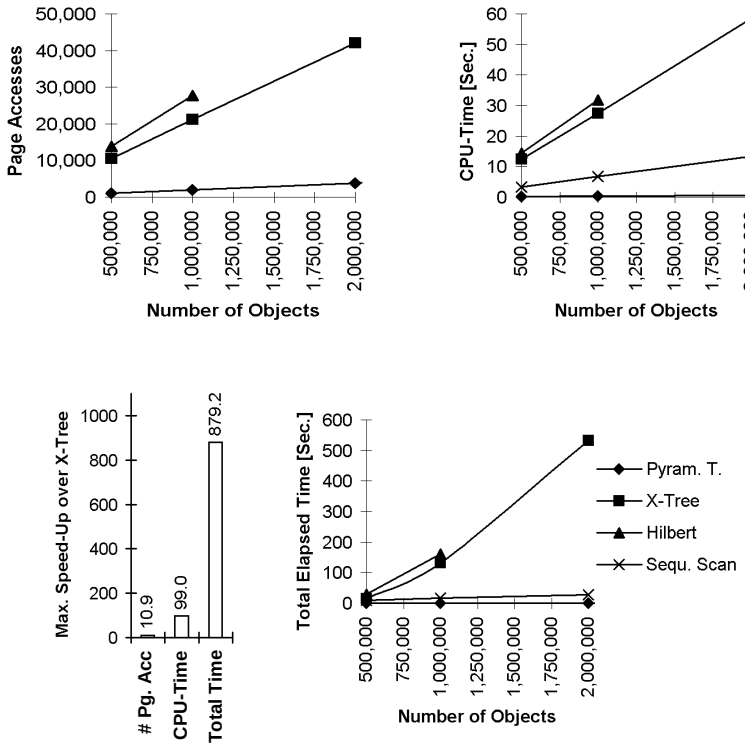


Figure 121: Performance Behavior over Database Size

Our evaluation comprises both, real and synthetic data sets. In all experiments, we performed range queries with a defined selectivity because range queries serve as a basic operation for other queries such as nearest neighbor queries or partial range queries. The query rectangles are selected randomly from the data space such that the distribution of the queries equals the distribution of the data set itself and the query rectangles are fully included in the data space. Thus, in case of uniform data we used uniformly distributed hypercube shaped query rectangles.

8.6.1 Evaluation Using Synthetic Data

Our synthetic data set contains 2,000,000 uniformly distributed points in a 100-dimensional data space. The raw data file occupies 800 MBytes of disk space. The main advantage of uniformly distributed point sets is that it is possible to scale down the dimensionality of the point set by projecting out some of the dimensions without affecting the semantics of the query. We created files with varying dimension and varying number of objects by projection and selection and constructed various indexes using these raw data files.

In our first experiment (cf. figure 121) we measured the performance behavior with varying number of objects. We performed range queries with 0.1% selectivity in a 16-dimensional data space and varied the database size from 500,000 to 2,000,000 objects. Unfortunately, using our implementation, the Hilbert-R-tree could only be constructed for a maximum of 1,000,000 objects due to the limited main memory. The file sizes of all indexes in this experiment sum up to 1.1 GigaBytes. The page size in this experiment was 4,096 Bytes, leading to an effective page capacity of 41.4 objects per page in all index structures. Figure 121 shows the performance of query processing in terms of number of page accesses, absorbed CPU-time and finally the total elapsed time, comprising CPU-time and time spent in disk I/O. The speed-up with respect to the number of page accesses seems to be almost constant and ranges between 9.78 and 10.91. The speed-up in CPU-time is higher than the speed-up in page accesses, but is only slightly increasing with growing database sizes. The reason is that B^+-trees facilitate an efficient in-page search for matching objects by applying bisection or interval search algorithms. However, most important is the speed-up in total elapsed time. It starts with factor 53, increases quickly and reaches its highest value with the largest database: The Pyramid-Technique with 2 million objects performs range queries 879 times faster than the corresponding X-tree! Range query processing on B^+-trees can be performed much more efficient than on X-trees because large parts of the tree can be traversed efficiently by following the side links in the data pages. Moreover, long-distance seek operations inducing expensive disk head movements have a lower probability due to better disk clustering possibilities in B^+-trees. The bar diagram on the right side of figure 121 summarizes the highest speed-up factors in this experiment.

In a second experiment, visualized in figure 122, we determined the influence of the data space dimension on the performance of query processing. For this purpose we created 5 data files as projections of the original data files with the dimensionalities 8, 12,

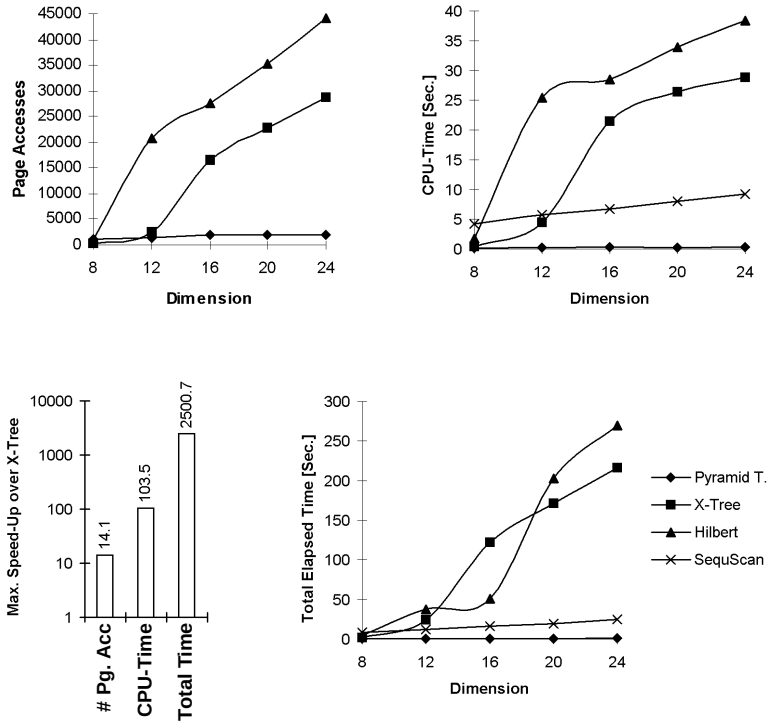


Figure 122: Performance Behavior over Data Space Dimension

16, 20, and 24 (the database size in this experiment is 1,000,000 objects) and created the corresponding indexes. The total amount of disk space occupied by the index structures used in this experiment sums up to 1.6 GigaBytes. The page size in this experiment was again 4,096 Bytes. The effective data page capacity depends on the dimension and ranged from 28 to 83 objects per page. We investigated range queries with a constant selectivity of 0.01%. For a constant selectivity, the query range varies according to the data space dimension.

We observed that the efficiency of query processing using the X-tree rapidly decreases with increasing dimension up to the point where large portions of the index are completely scanned (16-dimensional data space). From this point on, the page accesses are growing linearly with the index size. Even worse is the performance of the Hilbert R-

tree. A comparable deterioration of the performance with increasing dimension is not observable when using the Pyramid-Technique. Here, the number of page accesses, the CPU-time and total elapsed time grow slower than the size of the data set. The percentage of accessed pages with respect to all data pages is even reduced with growing dimensions (decreasing from 7.7% in the 8-dimensional experiment to 5.1% in the 24-dimensional experiment). The experiment yields a speed-up factor over the X-tree of up to 14.1 for the number of page accesses, and 103.5 for the CPU-time. Furthermore, the Pyramid-Technique is up to 2,500.7 times faster in terms of total elapsed time than the X-tree.

To demonstrate this observation that the percentage of pages accessed by the Pyramid-Technique decreases when going to higher dimensions, we determined the percentage of data pages accessed during query processing when indexing very high dimensions. Figure 123 depicts the result of this experiment: The percentage drops from 8.8% in 20 dimensions to 8.0% in 100 dimensions.

8.6.2 Evaluation Using Real Data Sets

In this series of experiments, we used data sets from two different application domains, information retrieval and data warehousing to demonstrate the practical impact of our technique.

The first data set contains text descriptors, describing substrings from a large text database extracted from WWW-pages. These text descriptors have been converted into 300,000 points in a 16-dimensional data space and were normalized to the unit hypercube. We varied the selectivity of the range queries from 10^{-5} to 31% and measured the

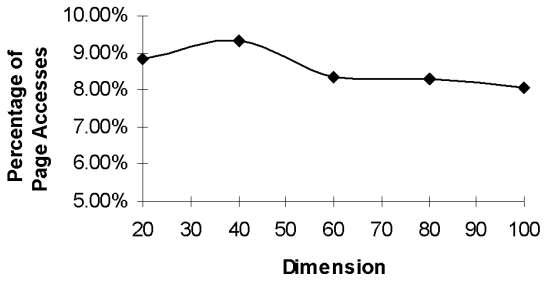


Figure 123: Percentage of Accessed Pages

query execution time (total elapsed time). The result is presented in figure 124 and confirms our earlier results on synthetic data that the Pyramid-Technique clearly outperforms the other index structures. The highest speed-up factor observed was 51. Additionally, the experiment shows that the Pyramid-Technique outperforms the competitive structures for any selectivity, i.e. for very small queries as well as for very large queries.

In a last series of experiments, we analyzed the performance of the Pyramid-Technique on a data set taken from a real-life data warehouse. The relation we used has 13 attributes: 2 categorical, 5 integer, and 6 floating point attributes. There are some very strong correlations in some of the floating point attributes, some of the attributes follow a very skewed distribution, whereas some other attributes are rather uniformly distributed. The actual data set we used comprises a subset of 803,944 tuples containing data of a few months. In a first experiment, we measured the real time consumed during query processing. Again, the Pyramid-Technique outperformed the other index structures by orders of magnitude. As expected, the speed-up increases when going to higher dimensions because the effects described in section 8.4 apply more for larger query ranges. However, even for the smallest query range in the experiment, the speed-up factor over the X-tree was about 10.47, whereas the speed-up for the largest query range was about 505.18 in total query execution time.

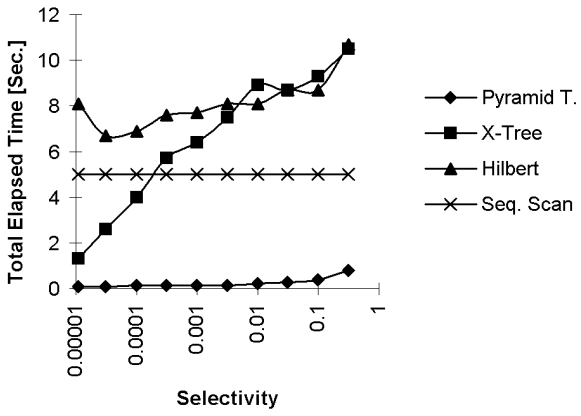


Figure 124: Query Processing on Text Data

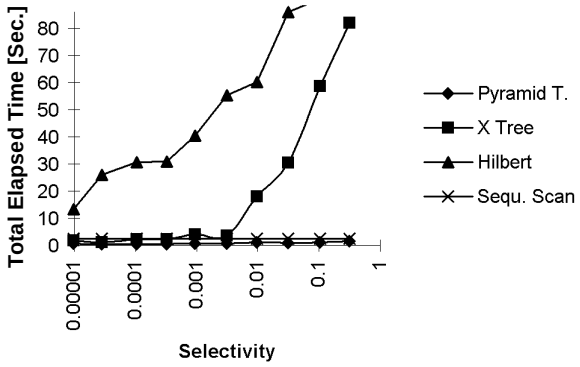


Figure 125: Query Processing on Warehousing Data

In a second experiment, we measured the effect of the extension of the Pyramid-Technique proposed in section 8.5. We made the experiment on this data set because the data is very skew and the median is rather close to the origin of the data space in most of the dimensions. Figure 125 shows the effect of the extension. For all selectivities, there was a speed-up of about 10-40%. This shows first that for very skewed data, it is worth to reorganize the index, and second that if we refuse to do so, the loss of performance is not too high compared to the high speed-up factors over other index structures.

A major point of criticism is the argument that the Pyramid-Technique is designed for hypercube shaped range queries and might perform bad for other queries. Therefore, we ran an additional experiment investigating the behavior of the Pyramid-Technique for skewed queries. We generated partial range queries shrinking the data space in k dimensions and having the full extension of the data space in (d-k) dimensions. These queries can be considered as (d-k)-dimensional hyper-slices in a d-dimensional space. As figure 126 shows, the Pyramid-Technique outperforms the linear scan for all of these queries except the 1-dimensional queries. For 1-dimensional queries, the Pyramid-Technique required 2.6 sec. compared to 2.48 sec. for the linear scan. However, a large improvement was observed for 8-dimensional to 13-dimensional queries. The X-Tree could not compete with the Pyramid-Technique for any of these queries.

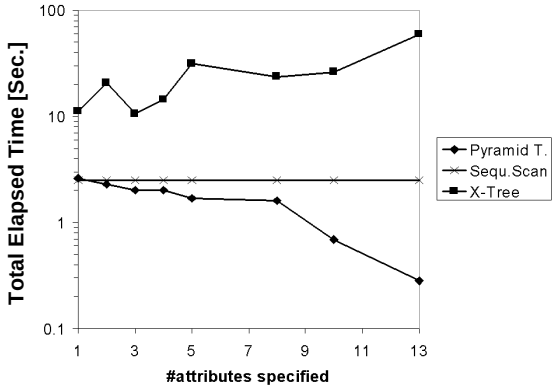


Figure 126: Varying the Query Mix (Warehouse Data)

Summarizing the results of our experiments, we make the following observations:

1. For nearly hypercube shaped queries, the Pyramid-Technique outperforms any competitive technique, including the linear scan. This holds even for skewed, clustered and categorical data.

2. For queries having a bad selectivity, i.e. a high number of answers or extremely skewed queries, especially queries specifying only a small number of attributes, the Pyramid-Technique still outperforms competitive index structures. A linear scan of the database, however, is faster.

Chapter 9
Conclusions

Indexing high-dimensional data spaces is an emerging domain of research. The material presented in this thesis has matured this new area both theoretically as well as practically. The theoretical contribution is in particular the part about cost models which is intended to bring deep insights into the problems and effects occurring in high-dimensional data spaces. Practical contributions are various new index structures and optimization techniques for high-dimensional data spaces.

9.1 Background

High-dimensional indexing is motivated by the similarity search problem in applications such as multimedia, CAD, medical image processing, molecular biology and time sequence analysis. For a similarity search, usually a so-called feature-transformation is applied. The *feature approach* extracts important properties from the objects in the database and transforms the objects into points of a high-dimensional vector space. Multidimensional index structures are applied for the management of these feature vectors.

Unfortunately, neither standard index structures and query processing techniques, nor the state-of-the-art in specialized index structures for high-dimensional data spaces yields satisfactory performance. Often the performance deteriorates when approaching dimensions higher than 20. The aim of this thesis was to overcome this drawback.

9.2 Contributions

For this purpose, in the beginning of this thesis (chapter 3), a cost model for query processing in high-dimensional data spaces was developed. We paid particular attention to boundary effects which occur in high-dimensional query processing. We also considered correlation effects which are inherent to data from real applications in contrast to artificial data from a uniform and independent distribution. Our cost model is applicable for query processing using both, the Euclidean metric and the maximum metric. It provides accurate estimates for the number of page accesses when executing range queries or nearest neighbor queries.

Based on this cost model, a number of optimization techniques for high-dimensional query processing was proposed in the rest of this thesis. We started (chapter 4) with an optimization of the logical block size of the index structure which is of particular importance in high-dimensional query processing. In our approach, the blocksize is adapted dynamically and independently in all pages to consider that the optimum may change when the database size increases.

Our next optimization technique is concerned with the dimension of the data space. On the one hand, our motivation was to overcome the deterioration of the performance of multidimensional index structures when moving to high-dimensional data spaces. On the other hand, the idea was inspired from the inverted-list-approach which builds a separate index on each attribute, merging the results of the single indexes for query processing. This approach, however, suffers from severe performance problems, because the merging step becomes too expensive if too many answers must be merged. The general idea of our approach is not to use one-dimensional indexes for each of the attributes but rather to decompose the vectors into sub-vectors of a moderate dimensionality to avoid the problems of both approaches, the high-dimensional index and the inverted-list-approach. The optimization task tackled in chapter 5 was therefore the suitable assignment of attributes to indexes.

Chapter 6 was devoted to the optimization of the shapes of the page regions in high-dimensional index structures. Whereas low-dimensional index structures tend to optimize for cube-like minimum bounding rectangles, we can derive from our cost model that this approach is inappropriate in the high-dimensional case. For high-dimensional query processing, an approach cutting thin slices from the data space boundary outperforms the classical approach of balanced splitting by large factors. We described this

strategy in context of a fast bulk-loading algorithm for the X-tree. This algorithm is also novel and a further research contribution of chapter 6.

In chapter 7, we proposed to exploit parallelism for high-dimensional query processing. The central task from a database point of view is the assignment of data to different servers. For this purpose, we developed a novel declustering method which was shown to be optimal with respect to our problem formalization. The idea is to decompose the data space into quadrants and to assign the quadrants to servers such that neighbors are assigned to different servers. The problem is equivalent to a special case of the graph coloring problem, but fortunately can be solved efficiently.

In chapter 8, we finally presented the *Pyramid-Technique*, an index which is highly adapted to processing of range queries using the maximum metric. The Pyramid-Technique maps the data points into a one-dimensional data space. This one-dimensional space can be indexed using conventional index structures such as the B^+-tree. We gain advantages such as an easy implementation within a commercial database system and the availability of sophisticated concurrency and recovery control mechanisms. The most important advantage of the Pyramid-Technique is, however, that query processing in the above mentioned cases is not subject to the so-called ‘curse of dimensionality’. The performance of query processing does not deteriorate when approaching higher dimensions.

9.3 Future Work

There are three major directions on which we will focus our future research: First, we will open new application domains to our techniques. Examples for promising application domains include biometrical data such as face recognition, fingerprints, voice identification, etc. The new challenge in these applications is the representation of uncertainty which is individual to single attributes. Few previous work exists for questions about the impact of uncertainty on multi-step query processing architectures and efficient indexing techniques.

A second direction of our future research is to integrate the refinement cost into our cost model and in the optimization process. There is a clear trade-off between index cost and refinement cost when performing, for instance dimension reduction or data space quantization, because these techniques improve the efficiency of index structures, but worsen the selectivity of the filter step. In our optimization techniques presented in the

current thesis, the relative selectivity of the filter step was consistently held fixed. A suitable application of reduction techniques, however, seems to be crucial for efficient query processing. Although optimal reduction is a challenging problem, few work exists in this area. We expect to automatize an important step in the chain of optimization for query processing.

Our third issue of future research concerns the practical applicability of high-dimensional indexes and the multi-step paradigm of query processing within a given information infrastructure. Most current work on multidimensional index structures and query processing in non-standard database applications is based on file system implementations and neglects issues such as data independence, concurrency and recovery. In contrast, most industrial companies have an information infrastructure based on a commercial (relational) database management system. The major database vendors extend the capabilities of their systems in the so-called object-relational approach trying to combine the advantages of relational and object-oriented database systems. Object-relational databases enable the application-specific implementation of index structures. In our future research, we will tackle the problem, how to integrate new indexing techniques into relational and object-relational database systems and, therefore, bridge one of the largest gaps in the practical applicability of the whole research direction of multidimensional query processing.

References

[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, LNCS 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, pp. 94-105 ,1998.

[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. of the 21st Conf. on Very Large Databases, 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, pp. 336-344, 1995.

[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 Databases (VLDB'91), Barcelona, Catalonia, 1991, pp. 81-90.

[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, Arizona, pp. 1-12, SIGMOD BEST PAPER AWARD.

[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.

[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, pp. 142-153,1998.

[BBK+ 98] Berchtold S., Böhm C., Keim D., Kriegel H.-P., Xu X.:*'Optimal Multidimensional Query Processing Using Tree Striping'*, submitted.

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

[BEK+ 98] Berchtold S., Ertl B., Keim D., Kriegel H.-P., Seidl T.: '*Fast Nearest Neighbor Search in High-Dimensional Spaces*', Proc. 14th Int. Conf. on Data Engineering, Orlando, 1998.

[Ben 75] Bentley J.L.: '*Multidimensional Search Trees Used for Associative Searching*', Communications of the ACM, Vol. 18, No. 9, pp. 509-517, 1975.

[Ben 79] Bentley J. L.: '*Multidimensional Binary Search in Database Applications'*, IEEE Trans. Software Eng. 4(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'*. Proceedings of 21th International Conference on Very Large Data Bases, VLDB'95, Zurich, Switzerland, 1995, pp. 299-310.

[BGRS 98] Beyer K., Goldstein J., Ramakrishnan R., Shaft U..: '*When Is "Nearest Neighbor" Meaningful?'*, submitted for publication, 1998.

[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, New York, pp. 139-143, 1998.

[BK 97] Berchtold S., Kriegel H.-P.: '*S3: Similarity Search in CAD Database Systems*', Proc. ACM SIGMOD Int. Conf. on Management of Data, 1997, Tucson, Arizona, pp. 564-567.

[BKK 96] Berchtold S., Keim D., Kriegel H.-P.: '*The X-Tree: An Index Structure for High-Dimensional Data*', 22nd Conf. on Very Large Databases, 1996, Bombay, India, pp. 28-39.

[BKK 97] Berchtold S., Keim D., Kriegel H.-P.: '*Using Extended Feature Objects for Partial Similarity Retrieval*', VLDB Journal Vol. 6, No. 4, pp. 333-348, 1997.

[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.

[BO 97] Bozkaya T., Ozsoyoglu M., *'Distance-Based Indexing for High-Dimensional Metric Spaces'*, Proc. 1997 ACM SIGMOD International Conference on Management of Data, Tucson, AZ, 1997.

[BM 77] Bayer R., McCreight E.M.: *'Organization and Maintenance of Large Ordered Indices'*, Acta Informatica 1(3), 1977, pp. 173-189.

[Bri 95] Brin S., *'Near Neighbor Search in Large Metric Spaces'*, Proc. 21st VLDB Conference, 1995, pp. 574-584.

[BSW 97] van den Bercken J., Seeger B., Widmayer P.:, *'A General Approach to Bulk Loading Multidimensional Index Structures'*, 23rd Conf. on Very Large Databases, 1997, Athens, Greece.

[CD 97] Chaudhuri S., Dayal U.: *'Data Warehousing and OLAP for Decision Support'*, Tutorial, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1997, Tucson, Arizona.

[Chi 94] Chiueh T., '*Content-Based Image Indexing*', Proc. 20th VLDB Conference, 1994, pp. 582-593.

[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, pp. 183-192, 1979.

[Com 79] Comer D.: *'The Ubiquitous B-tree'*, ACM Computing Surveys 11(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 Databases (VLDB'97), Athens, Greece, 1997.

[DH 73] Duda R. O., Hart P. E.: '*Pattern Classification and Scene Analysis'*, Wiley, New York, 1973.

[DS 82] Du H.C., Sobolewski J.S.: '*Disk allocation for cartesian product files on multiple Disk systems*', ACM TODS, Journal of Transactions on Database Systems, 1982, pp. 82-101.

[Eas 81] Eastman C.M.: '*Optimal Bucket Size for Nearest Neighbor Searching in k-d Trees'*, Information Processing Letters Vol. 12, No. 4, 1981.

[Eva 94] Evangelidis G.: *'The hB$^{\pi}$-Tree: A Concurrent and Recoverable Mult-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, 1985, pp. 227-238.

[Fal 88] Faloutsos C.: *'Gray Codes for Partial Match and Range Queries'*, IEEE Trans. on Software Engineering 14, 1988, pp. 1381-1393.

[FB 74] Finkel R, Bentley J.L. *'Quad Trees: A Data Structure for Retrieval of Composite Keys'*, Acta Informatica 4(1), 1974, pp. 1-9.

[FB 93] Faloutsos C., Bhagwat P.: '*Declustering Using Fractals*', 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 Transactions on Mathematical Software, Vol. 3, No. 3, September 1977, pp. 209-226.

[FBFH 94] Faloutsos C., Barber R., Flickner M., Hafner J., et al.: *'Efficient and Effective Querying by Image Content'*, Journal of Intelligent Information Systems, 1994, Vol. 3, pp. 231-262.

[FG 96] Faloutsos C., Gaede V.: *'Analysis of n-Dimensional Quadtrees using the Hausdorff Fractal Dimension'*, Proceedings of 22th International Conference 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'*, Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Minneapolis, Minnesota, 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.

[FRM 94] Faloutsos C., Ranganathan M., Manolopoulos Y.: *'Fast Subsequence Matching in Time-Series Databases'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 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, 1987.

[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 Symposium on Advances in Spatial Databases, SSD'95, Portland, Maine, USA, 1995, Lecture Notes in Computer Science Vol. 951, pp. 96-116.

[Gar 82] Gargantini I.: *'An Effective Way to Represent Quadtrees'*, Comm. of the ACM, Vol. 25, No. 12, 1982, pp. 905-910.

[GG 98] Gaede V., Günther O.: *'Multidimensional Access Methods'*, 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 Eng, 1989.

[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'*, Proceedings of the 2nd ACM Workshop on Advances in Geographic Information Systems, ACM Press, Gaithersburg, Maryland, pp. 136-143, 1994.

[Hen 98] Henrich, A.: '*The LSD^h-tree: An Access Structure for Feature Vectors*', Proc. 14th Int. Conf. on Data Engineering, Orlando, 1998.

[Hin 85] Hinrichs K.: *'Implementation of the Grid File: Design Concepts and Experiance'*, BIT 25, pp. 569-592.

[Hoa 62] C.A.R. Hoare, *'Quicksort'*, 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, Portland, ME, 1995, pp. 83-95.

[HSW 88a] Hutflesz A., Six H.-W., Widmayer P.: *'Globally Order Preserving Multidimensional Linear Hashing'*, Proc. 4th IEEE Int. Conf. on Data Eng., 1988, pp. 572-579.

[HSW 88b] Hutflesz A., Six H.-W., Widmayer P.: *'Twin Grid Files: Space Optimizing Acces Schemes'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1988.

[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.

[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, 1991, pp. 208-217.

[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 Databases, 1994, pp. 500-509.

[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.

[Knu 75] Knuth D. E.: *'The Art of Computer Programming'*, Volume 3, Addison-Wesley, Reading, MA, 1975.

[Kor+ 96] Korn F., Sidiropoulos N., Faloutsos C., Siegel E., Protopapas Z.: *'Fast Nearest Neighbor Search in Medical Image Databases'*, Proc. 22nd VLDB Conference, Mumbai (Bombay), India, 1996, pp. 215-226.

[KP 88] Kim M.H., Pramanik S.: '*Optimal file distribution for partial match retrieval*', Proc. ACM SIGMOD Int. Conf. on Management of Data, 1988, pp. 173-182.

[Kri 84] Kriegel H.-P.: *'Performance Comparison of Index Structures for Multi-Key Retrieval'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Boston, MA, 1984, pp. 186-196.

[KS 86] Kriegel H.-P., Seeger B.: *'Multidimensional Order Preserving Linear Hashing with Partial Extensions'*, Proc. Int. Conf. on Database Theory, in: Lecture Notes in Computer Science, Vol. 243, Springer, 1986.

[KS 87] Kriegel H.-P., Seeger B.: *'Multidimensional Dynamic Quantile Hashing is very Efficient for Non-Uniform Record Distributions'*, Proc 3rd Int. Conf. on Data Engineering, 1987, pp. 10-17.

[KS 88] Kriegel H.-P., Seeger B.: *'PLOP-Hashing: A Grid File Without Directory'*, Proc. 4th Int. Conf. on Data Engineering, 1988, pp. 369-376.

[KS 89] Kriegel H.-P., Seeger B.: *'Multidimensional Quantile Hashing Is Very Efficient for Non-Uniform Distributions'*, Information Sciences 48, 1989, pp. 99-117.

[KS 97] 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, 1997, pp. 369-380.

[KS 98] Kriegel H.-P., Seidl T.: '*Approximation-Based Similarity Search for 3-D Surface Segments*', GeoInformatica Journal, Kluwer Academic Publishers, 1998, to appear.

[KSS 97] Kriegel H.-P., Schmidt T., Seidl T.: *'3D Similarity Search by Shape Approximation'*, Proc. Fifth Int. Symposium on Large Spatial Databases (SSD'97), Berlin, Germany, 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'*, VLDB Journal, Vol. 3, pp. 517-542, 1995.

[Lum 70] Lum, V.Y.: *'Multi-attribute Retrieval with Combined Indexes'*, Communications of the ACM, Vol. 13, 11, November, 1970, pp. 660-665.

[LS 89] Lomet D., Salzberg B.: *'The hB-tree: A Robust Multiattribute Search Structure'*, Proc. 5th IEEE Int. Conf. on Data Eng., 1989, pp. 296-304.

[LS 90] Lomet D., Salzberg B.: *'The hB-tree: A Multiattribute Indexing Method with Good Guaranteed Performance'*, ACM Trans. on Data Base Systems 15(4), 1990, pp. 625-658.

[Man 77] Mandelbrot B.: '*Fractal Geometry of Nature*', W. H. Freeman and Company, New York, 1977.

[MG 93] Mehrotra R., Gary J.: *'Feature-Based Retrieval of Similar Shapes'*, Proc. 9th Int. Conf. on Data Engeneering, April 1993

[MG 95] Mehrotra R., Gary J.: *'Feature-Index-Based Sililar Shape retrieval'*, Proc. of the 3rd Working Conf. on Visual Database Systems, March 1995

[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', Communications of the ACM, Vol. 14, 12, December, 1971, pp. 775-776.

[NHS 84] Nievergelt J., Hinterberger H., Sevcik K. C.: *'The Grid File: An Adaptable, Symmetric Multikey File Structure'*, 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, pp. 150-157, 1982.

[Ore 90] Orenstein J., : *'A comparison of spatial query processing techniques for native and parameter spaces'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1990, pp. 326-336.

[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, 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.

[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, pp. 394-408, 1997.

[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'*, Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS'93, Washington, D.C., 1993, pp.214-221.

[RKV 95] Roussopoulos N., Kelley S., Vincent F.: '*Nearest Neighbor Queries*', Proc. ACM SIGMOD Int. Conf. on Management of Data, 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, 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 Transactions on Signal Processing, Vol. 40, No. 3, March 1992, pp. 518-531.

[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.

[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.

[SH 94] Shawney H., Hafner J.: '*Efficient Color Histogram Indexing*', Proc. Int. Conf. on Image Processing, 1994, pp. 66-70.

[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, 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.

[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, pp. 579-589, 1991.

[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 Databases, Brighton, England, 1987, pp. 507-518.

[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] Yannis Theodoridis, Timos K. Sellis: *'A Model for the Prediction of R-tree Performance'*. Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, 1996, Montreal, Canada. ACM Press, 1996, ISBN 0-89791-781-2 pp. 161-171.

[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,Compute Science Press, Rockville, MD, 1989.

[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, New Orleans, LA, 1996.

[WSB 98] Weber R., Schek H.-J., Blott S.: '*A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces*', Proc. Int. Conf. on Very Large Databases, New York, 1998.

[WW 80] Wallace T., Wintz P.: *'An Efficient Three-Dimensional Aircraft Recognition Algorithm Using Normalized Fourier Descriptors'*, Computer Graphics and Image Processing, Vol. 13, pp. 99-126, 1980.

[Yia 93] Yiannilos P. N., '*Data Structures an Algorithms for Nearest Neighbor Search in General Metric Spaces'*, ACM-SIAM Symposium 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.

Index

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Curriculum Vitae

Christian Böhm was born on September 28, 1968 in Rosenheim, Germany. After visiting primary school from 1975 to 1979 he attended secondary school from 1979 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., chair for database and knowledge base systems at the *TUM*, and by Professor Dr. J. Christoph Freytag and Dr. Frank Schönefeld at the database systems research group of *Digital Equipment* (*DEC*).

In July 1994, he entered the research group for knowledge bases of the *FORWISS* institute (*Bayerisches Forschungszentrum für wissensbasierte Systeme*) which is supervised by Professor R. Bayer, Ph.D. Christian Böhm 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*. He received the SIGMOD Best-Paper-Award 1997 for a joint publication with Dr. Stefan Berchtold, Bernhard Braunmüller, Professor Dr. Daniel Keim and Professor Dr. Hans-Peter Kriegel.