First publ. in: ACM computing surveys 33 (2001), 3, pp. 322-373

Searching in High-dimensional Spaces - Index Structures for Improving the Performance of Multimedia Databases

Christian Böhm
University of Munich
Germany
boehm@informatik.uni-muenchen.de

Stefan Berchtold
stb gmbh
Germany
Stefan.Berchtold@stb-gmbh.de

Daniel A. Keim
University of Halle
Germany
keim@informatik.uni-halle.de

Abstract

During the last decade, multimedia databases have become increasingly important in many application areas such as medicine, CAD, geography, or molecular biology. An important research issue in the field of multimedia databases is the content based retrieval of similar multimedia objects such as images, text, and videos. However, in contrast to searching data in a relational database, a content based retrieval requires the search of similar objects as a basic functionality of the database system. Most of the approaches addressing similarity search use a so-called feature transformation which transforms important properties of the multimedia objects into high-dimensional points (feature vectors). Thus, the similarity search is transformed into a search of points in the feature space which are close to a given query point in the high-dimensional feature space. Query Processing in high-dimensional spaces has therefore been a very active research area over the last few years. A number of new index structures and algorithms have been proposed. It has been shown that the new index structures considerably improve the performance in querying large multimedia databases. Based on recent tutorials [BK 98, BK 00], in this survey we provide an overview of the current state-of-the-art in querying multimedia databases, describing the index structures and algorithms for an efficient query processing in high-dimensional spaces. We identify the problems of processing queries in high-dimensional space, and we provide an overview of the proposed approaches to overcome these problems.

1. Indexing Multimedia Databases

Multimedia databases are of high importance in many application areas such as geography, CAD, medicine, or molecular biology. Depending on the application, the multimedia databases need to have different properties and need to support different types of queries. In contrast to traditional database applications, where point, range, and partial match queries are very important, multimedia databases require a search for all objects in the database which are similar (or complementary) to a given search object. In the following, we describe the notion of similarity queries and the feature-based approach to process those queries in multimedia databases in more detail.

1.1 Feature-based Processing of Similarity Queries

An important aspect of similarity queries is the similarity measure. There is no general definition of the similarity measure since it depends on the needs of the application and is therefore highly application-dependent. Any similarity measure, however, takes two objects as input parameters and determines a positive real number, denoting the similarity of the two objects. A similarity measure is therefore a function of the form

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

In defining similarity queries, we have to distinguish between two different tasks, which are both important in multimedia database applications: ε*-similarity* means that we are interested in all objects of which the similarity to a given search object is below a given threshold ε, and *NN-similarity* means that we are only interested in the objects which are the most similar ones with respect to the search object.

Definition 1 *(*ε*-Similarity, Identity)*
Two objects obj_1 and obj_2 are called $\varepsilon - similar$ iff $\delta(obj_2, obj_1) < \varepsilon$.
(For $\varepsilon = 0$, the objects are called $identical$.)

Note that this definition is independent from database applications and just describes a way to measure the similarity of two objects.

Definition 2 *(NN-Similarity)*
Two objects obj_1 and obj_2 are called $NN - similar$ with respect to a database of objects DB iff $\forall obj \in DB, obj \neq obj_1: \delta(obj_2, obj_1) \leq \delta(obj_2, obj)$.

We are now able to formally define the ε*-similarity query* and the *NN-similarity query*.

Definition 3 *(*ε*-Similarity Query, NN-Similarity Query)*
Given a query object obj_s, find all objects obj from the database of objects DB which are $\varepsilon - similar$ ($identical$ for $\varepsilon = 0$) to obj_s, i.e. determine

$$\{obj \in DB \mid \delta(obj_s, obj) < \varepsilon\}.$$

Given a query object obj_s, find the object(s) $\overline{obj}$ from the database of objects DB which are $NN - similar$ to obj_s, i.e. determine

$$\{\overline{obj} \in DB \mid \forall obj \in DB, obj \neq \overline{obj}: \delta(obj_s, \overline{obj}) \leq \delta(obj_s, obj)\}.$$

The solutions which are currently used to solve similarity search problems are mostly feature-based solutions. The basic idea of feature-based similarity search is to extract important features from the multimedia objects, map the features into high-dimensional feature

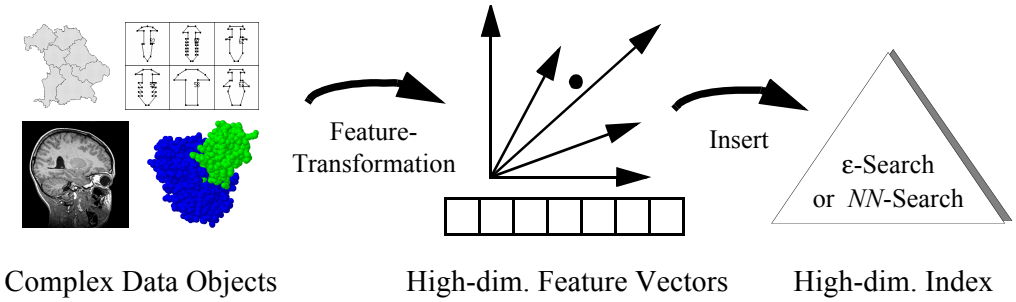


Figure 1: Basic Idea of Feature-based Similarity Search.

vectors, and search the database of feature vectors for objects with similar feature vectors (cf. figure 1). The *feature transformation F* is defined as the mapping of the multimedia object *(obj)* into a *d*-dimensional feature vector

$$F:\ Obj \rightarrow \Re^d .$$

The similarity of two objects obj_1 and obj_2 can now be determined

$$\delta(obj_1, obj_2) = \delta_{Euclid}(F(obj_1), F(obj_2)) .$$

Feature-based approaches are used in many application areas including molecular biology (for molecule docking) [SBK 92], information retrieval (for text matching) [AGMM 90], multimedia databases (for image retrieval) [FBFH 94, SK 97], sequence databases (for subsequence matching) [AFS 93, FRM 94, ALSS 95], geometric databases (for shape matching) [MG 93, MG 95, KSF+ 96] etc. Examples of feature vectors are color histograms [SH 94], shape descriptors [Mum 87, Jag 91, MG 95], Fourier vectors [WW 80], text descriptors [Kuk 92], etc. The result of the feature transformation are sets of high-dimensional feature vectors. The similarity search now becomes an ε–query or a nearest neighbor query on the feature vectors in the high-dimensional feature space, which can be handled much more efficiently on large amounts of data than the time-consuming comparison of the search object to all complex multimedia objects in the database. Since the databases are very large and consist of millions of data objects with several tens to a few hundreds of dimensions, it is essential to use appropriate multidimensional indexing techniques to achieve an efficient search of the data. Note that the feature transformation often also involves complex transformations of the multimedia objects such as a the feature extraction, normalization, or fourier transformation. Depending on the application, these operations may be necessary to achieve, for example, invariance with respect to a scaling or rotation of the objects. The details of the feature transformations are beyond the scope of this paper. For further reading on feature transformations, the interested reader is referred to the literature [SH 94, Mum 87, Jag 91, MG 95, WW 80, Kuk 92].

For an efficient similarity search it is necessary to store the feature vectors in a high-dimensional index structure and use the index structure to efficiently evaluate the distance metric. The high-dimensional index structure used must efficiently support

- point queries for processing $identity$ queries on the multimedia objects,
- range queries for processing $\varepsilon - similarity$ queries, and
- nearest neighbor queries for processing $NN - similarity$ queries.

Note that instead of using a feature transformation into a vector space, the data can also be directly processed using a metric space index structure. In this case, the user has to provide a metric which corresponds to the properties of the similarity measure. The basic idea of metric indexes is to use the given metric properties to build a tree which then can be used to prune branches in processing the queries. The basic idea of metric index structures will be discussed in section 5. A problem of metric indexes is that they only use information about the data than vector space index structures which results in poorer pruning and also a poorer performance than vector space index structures. A nice possibility to improve this situation is the FASTMAP algorithm [FL 95] which maps the metric data into a lower-dimensional vector space and uses a vector space index structure for an efficient access to the transformed data.

Due to their practical importance, in this survey we restrict ourselves to vector space index structures. We assume to have some given application-dependent feature transformation which provides a mapping of the multimedia objects into some high-dimensional space. There is a quite large number of index structures which have been developed for an efficient query processing in some multidimensional space. In general, the index structures can be classified in two groups: Data organizing structures such as R-trees [Gut 84, BKSS 90] and space organizing structures such as Multidimensional Hashing [Oto 84, KS 86, KS 87, KS 88, SK 90], GRID-Files [NHS 84, Fre 87, Hin 85, KW 85, KS 88, Ouk 85, HSW 88b], and kd-tree-based methods (k-d-B-tree [Rob81], hB-tree [LS89, LS90, Eva94], and LSD-h-tree [Hen98]).

For a comprehensive description of most multidimensional access methods, primarily concentrating on low-dimensional indexing problems, the interested reader is referred to a recent survey by Gaede and Guenther [GG 98]. The survey by Gaede and Guenther, however, does not tackle the problem of indexing multimedia databases which requires an efficient processing of nearest-neighbor queries in high-dimensional feature spaces; and therefore, the survey does not deal with nearest-neighbor queries and the problems of indexing high-dimensional spaces. In our survey, we focus on the index structures which have been specifically designed to cope with the effects occurring in high-dimensional space. Since hashing-

and GRID-File-based methods do not play an important role in high-dimensional indexing, we do not cover them in the survey.[1] The reason why hashing techniques are not used in high-dimensional spaces are the problems that arise in high-dimensional space. To be able to understand these problems in more detail, in the following we discuss some general effects that occur in high-dimensional spaces.

1.2 Effects in High-dimensional Space

A broad variety of mathematical effects can be observed when one increases the dimensionality of the data space. Interestingly, some of these effects are not of quantitative nature but of qualitative nature. In other words, one cannot think about these effects by simply extending 2- or 3-dimensional experiences. Rather, one has to think e.g. at least 10-dimensional to even see the effect occurring. Furthermore, some are pretty non-intuitive. Few of the effects are of pure mathematical interest while some others have severe implications to the performance of multi-dimensional index structures. Therefore, in the database world, these effects are subsumed by the term "curse of dimensionality". Generally speaking, the problem is that important parameters such as volume and area depend exponentially from the number of dimensions of the data space. Therefore, most index structures proposed so far operate efficiently only if the number of dimensions is fairly small. The effects are non-intuitive because we are used to deal with 3-dimensional spaces in the real world but these effects do not occur in low-dimensional spaces. Many people even have trouble understanding spatial relations in 3-dimensional spaces, however, no one can "imagine" a 8-dimensional space. Rather, we always try to find a low-dimensional analogy when dealing with such spaces. Note that there actually is no notion of a "high"-dimensional spaces. There is only higher and lower. Nevertheless, if people talk about high-dimensional, they usually mean a dimension of about 10 to 16, or at least 5 or 6.

Next, we will list the most relevant effects and try to classify them:

- pure geometric effects concerning the surface and volume of (hyper-) cubes and (hyper-) spheres
  - •the volume of a cube grows exponentially with increasing dimension (and constant edge length)
  - •the volume of a sphere grows exponentially with increasing dimension
  - •most of the volume of a cube is very close to the (d-1)-dimensional surface of the cube.

1. The only exception to this is a technique for searching approximate nearest neighbors in high-dimensional spaces which has been proposed in [GIM 99].

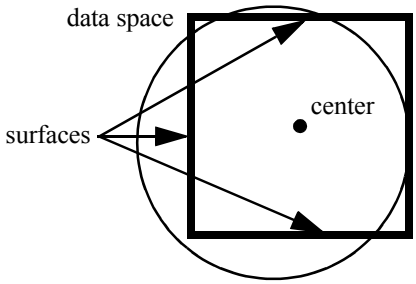


Figure 2: Spheres in high-dimensional spaces.

- effects concerning the shape and location of index partitions
  - •a typical index partition in high-dimensional spaces will span the majority of the data space in most dimensions and only be split in a few dimensions
  - •a typical index partition will not be cubic, rather it will "look" like a rectangle
  - •a typical index partition touches the boundary of the data space in most dimensions
  - •the typical partitioning of space is coarser the higher the dimension
- effects arising in a database environment (e.g. selectivity of queries)
  - •assuming uniformity, a reasonably selective range query corresponds to a hyper-cube having a huge extension in each dimension
  - •assuming uniformity, a reasonably selective nearest-neighbor query corresponds to a hyper-sphere having a huge radius in each dimension. Usually, this radius is even larger than the extension of the data space in each dimension.

To be more precise, we will present some of the listed effects in more depth and detail in the rest of the section.

To demonstrate, how much we stick to our understanding of low-dimensional spaces, consider the following lemma: Consider a cubic-shaped d-dimensional data space of extension $[0, 1]^d$. We define the center-point c of the data space as the point (0.5, ..., 0.5). The lemma "Every d-dimensional sphere touching (or intersecting) the (d-1)-dimensional boundaries of the data space also contains c." is obviously true for d=2, as one can take from figure 2. Spending some more effort and thinking, we are able to also proof the lemma for d=3. However, the lemma is definitely false for d=16, as the following counter example shows. Define a sphere around the point p=(0.3, ... 0.3). This point p has a Euclidean distance of $\sqrt{d \cdot 0{,}2^2} = 0{,}8$ from the center point. If we define the sphere around p with a radius of 0.7, the sphere will touch (or intersect) all 15-dimensional surfaces of the space. However, the center point is not included in the sphere. We have to be aware of the fact that effects like this are not only nice mathematical properties but also lead to severe conclusions for the performance of index structures.

The most basic effect is the exponential growth of volume. The volume of a cube in a d-dimensional space is of the formula: $vol = e^d$, where d is the dimension of the data space and e is the edge length of the cube. Now if the edge length is a number between 0 and 1, the volume of the cube will exponentially decrease when increasing the dimension. Viewing the problem from the opposite side, if we want to define a cube of constant volume for increasing dimensions, the appropriate edge length will quickly approach 1. For example, in a 2-dimensional space of extension $[0, 1]^d$, a cube of volume 0.25 has an edge length of 0.5 whereas in a 16-dimensional space, the edge length has to be $\sqrt[16]{0{,}25} \approx 0{,}917$.

The exponential growth of the volume has a serious impact on conventional index structures. Space organizing index structures, for example, suffer from the "dead space" indexing problem. Since space organizing techniques index the whole domain space, a query window may overlap part of the space belonging to a page that actually contains no points at all.

Another important issue is the space partitioning one can expect in high-dimensional spaces. Usually, index structures split the data space using (d-1)-dimensional hyperplanes; for example, in order to perform a split, the index structure selects a dimension (the split dimension) and a value in this dimension (the split value). All data items having a value in the split dimension smaller than the split value are assigned to the first partition whereas the other data items form the second partition. This process of splitting the data space continues recursively until the number of data items in a partition is below a certain threshold and the data items of this partition are stored in a data page. Thus, the whole process can be described by a binary tree, the split tree. As the tree is a binary tree, the height h of the split tree usually depends logarithmically on the number of leaf nodes i.e. data pages. On the other hand, the number d' of splits for a single data page is on average

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

where N is the number of data items and $C_{eff}(d)$ [1] is the capacity of a single data page. Thus, we can conclude that if all dimensions are equally used as split dimensions, a data page has been split at most once or twice in each dimension and therefore, spans a range between 0.25 and 0.5 in each of the dimensions (for uniformly distributed data). From that, we may conclude that the majority of the data pages is located at the surface of the data space rather than in the interior. Additionally, this obviously leads to a coarse data space partitioning in single dimensions. However, from our understanding of index structures such as the R*-Tree which had been designed for geographic applications we are used to very fine partitions where the majority of the data pages is in the interior of the space. and we have to be careful

1. For most index structures, the capacity of a single data page depends on the dimensionality since the number of entries decreases with increasing dimension due to the larger size of the entries.

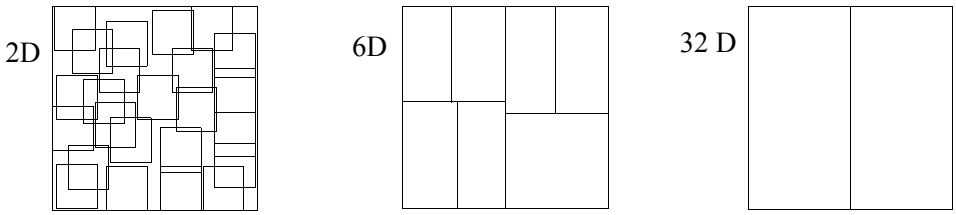


Figure 3: Space partitioning in high-dimensional spaces.

not to apply this understanding to high-dimensional spaces. Figure 3 depicts the different configurations. Note that this effect applies to almost any index structure proposed so far because we only made assumptions about the split algorithm.

Additionally, not only index structures show a strange behavior in high-dimensional spaces but also the expected distribution of the queries is affected by the dimensionality of the data space. If we assume a uniform data distribution, the selectivity of a query (the fraction of data items contained in the query) directly depends on the volume of the query. In case of nearest-neighbor queries, the query affects a sphere around the query point which exactly contains one data item, the NN-sphere. According to [BBKK 97], the radius of the NN-sphere increases rapidly with increasing dimension. In a data space of extension $[0, 1]^d$, it quickly reaches a value larger than 1 when increasing d. This is a consequence of the above mentioned exponential relation of extension and volume in high-dimensional spaces.

Considering all these effects, we can conclude that if one builds an index structure using a state-of-art split algorithm the performance will deteriorate rapidly when increasing the dimensionality of the data space. This has been realized not only in the context of multimedia systems where nearest-neighbor queries are most relevant, but also in the context of data warehouses where range queries are the most frequent type of query [BBK 98, BBK 98a]. Theoretical results based on cost models for index-based nearest neighbor and range queries also confirm the degeneration of the query performance [YY 85, BBKK 97, BBKK 00, BGRS 99]. Other relevant cost models proposed before include [FBF 77, Cle 79, Eas 81, Spr 91, PSTW 93, AMN 95, Ary 95, TS 96, PM 97a].

1.3 Basic Definitions

Before we are able to proceed, we need to introduce some notions and to formalize our problem description. We will define in this section 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.

1.3.1 Database

We assume that in our similarity search application, objects are feature-transformed into points of a vector space with 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 $\mathbb{R}^d$. 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 latter is the query point. From the context, the intended meaning of the notion point will always be obvious.

Definition 4 . Database

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

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

$$P_i \in DS, i = 0..n-1, DS \subseteq \mathbb{R}^d.$$

1.3.2 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 distances are applied. Most common is the Euclidean metric L_2 defining the usual Euclidean distance function:

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

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

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

Queries using the L_2 metric are (hyper-) sphere shaped. Queries using maximum metric or Manhattan metric are hypercubes and rhomboids, respectively (c.f. Figure 4). 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:

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

Arbitrarily rotated ellipsoids can be defined using a positive definite similarity matrix W. This concept is used for adaptable similarity search [Sei 97]:

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

1.3.3 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.

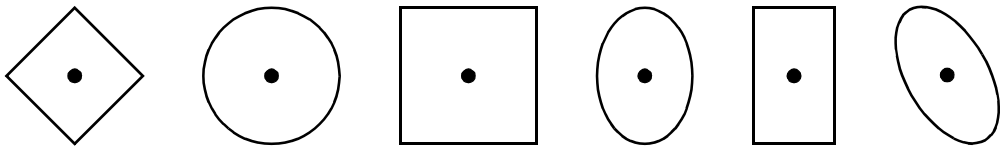


Figure 4: Metrics for Data Spaces.

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:

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

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, the object with the lowest distance to the query point among all points stored in the database[1]. 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:

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

1. A recent extension of nearest neighbor queries are closest pair queries which are also called distance joins [HS 98, CMTV 00]. This query type is mainly important in the area of spatial databases and therefore, closest pair queries are beyond the scope of this survey.

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

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

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:

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

A variant of k-nearest neighbor queries are **ranking queries** which do not require that the user specifies a range in the data space or a result set size. The first answer of a ranking query is always the nearest neighbor. Then, 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.

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 neighbor. 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.

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 sec-

ondary storage, all queries can be evaluated using a so-called *sequential scan*, which is faster than the access of small blocks spread over wide parts of 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. Note that we assume that there is no parallelism between CPU and disk I/O for any query processing technique presented in this paper.

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.

1.5 Overview

The rest of the survey is organized as follows: We start with describing the common principles of multidimensional index structures and the algorithms used to build the indexes and process the different query types. Then, we provide a systematic overview of the querying and indexing techniques which have been proposed for high-dimensional data spaces, describing them in a uniform way and discuss their advantages and drawbacks. Rather than describing the details of all the different approaches, we try to focus on the basic concepts and algorithms used. We also cover a number of recently proposed techniques dealing with optimization and parallelization issues. In concluding the survey, we try to stir up further research activities by presenting a number of interesting research problems.

2. Common Principles of High-Dimensional Indexing Methods

2.1 Structure

High-dimensional indexing methods are based on the principle of hierarchical clustering of the data space. Structurally, they resemble 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 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 and

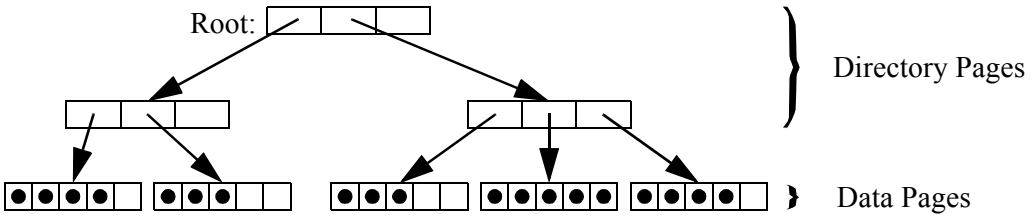


Figure 5: Hierarchical Index Structures.

consist of (key, pointer)-tuples. The key information is different for different index structures. For B-trees, for example, the keys are ranges of numbers and for an R-tree the keys are bounding boxes. There is a single directory node, which is called the root node. It serves as an entry point for query and update processing. The index structures are height-balanced. That means, the lengths of the paths between the root and all data pages are identical, but may change after insert or delete operations. The length of a path from the root to a data page is called the *height* of the index structure. 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.

The uniform (key, pointer)-structure of the directory nodes also allows an implementation of a wide variety of index structures as extensions of a generic index structure as done in the Generalized Search Tree [HNP 95]. The Generalized Search Tree (GiST) provides a nice framework for a fast and reliable implementation of search trees. The main requirement for defining a new index structure in GiST is to define the keys and provide an implementation of four basic methods which are needed for building and searching the tree (cf. section 3). Additional methods may be defined to enhance the performance of the index, which is especially relevant for similarity or nearest-neighbor queries [Aok 98]. An advantage of GiST is that the basic data structures and algorithms as well as main portions of the concurrency and recovery code can be reused. It is also useful as a basis for theoretical analysis of indexing schemes [HKP 97]. A recent implementation in a commercial object-relational system shows that GiST-based implementations of index structures can provide a competitive performance while considerably reducing the implementation efforts [Kor 99].

2.2 Management

The high-dimensional access methods are designed primarily for 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 chose $C_{max,data}$ and $C_{max,dir}$ such that data and directory nodes fit exactly into the pages of secondary stor-

age. However, in modern operating systems, the page size of a disk drive is considered as a hardware detail hidden from programmers and users. Despite that, 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 properly chose 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 (c.f. section 6.2). 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 creation of the index. This **D**ynamic **A**doption of the **B**lock **S**ize gives the DABS-tree [Boe 98] 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 ca. 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 analytically derive an average storage utilization, further on referred to as 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 usually has to be determined experimentally[1].

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

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

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

1. For the hB-tree, it has been shown in [LS 90] that under certain assumptions the average storage utilization is 67%.

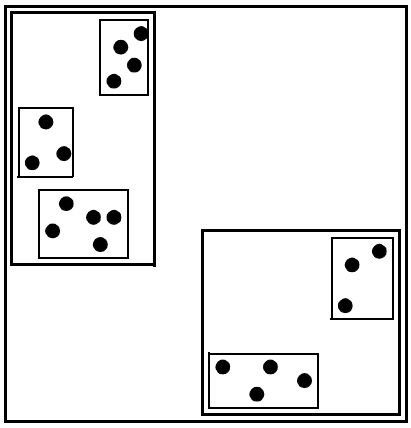


Figure 6: Corresponding Page Regions of an Indexing Structure.

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 several among the above. For most, but not all high-dimensional index structures, the page region is a contiguous, solid, 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 are avoided if possible or at least minimized.

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

In query processing, the page region is used to cut 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 referred to 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 worse efficiency in query processing, as we will see later.

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 using a specific indexing structure, they are presented here in a more general way.

3.1 Insert, Delete and Update

Insert, delete and update are the operations, which are most specific to the corresponding index structures. Despite that, there are basic algorithms capturing all actions which are common to all index structures. In the GiST framework [HNP 95], the build-up of the tree via the insert operation is handled using three basic operations: *Union*, *Penalty*, and *PickSplit*. The *Union* operation consolidates information in the tree and returns a new key which is true for all data items in the considered subtree. The *Penalty* operation is used to find the best path for inserting a new data item into the tree by providing a number representing how bad an insertion into that path would be. The *PickSplit* operations is used to split a data page in case of an overflow.

The insertion and delete operations of tree structures are usually the most critical operation which heavily determine the structure of the resulting index and the achievable performance. Some index structures require for a simple insert the propagation of changes towards the root or down the children as, for example, in case of the R-tree and k-d-B-tree and some that don't as, for example, the hB-tree. In the latter case, the insert/delete operations are

called *local* operations while in the first case, the are called *non-local* operations. 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 $C_{max,data}$, 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 $C_{max,dir}$, 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.

Heuristics individual to the specific indexing structure are applied for the following subtasks:

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

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

The choice of heuristics in 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 usually negatively affected.[1] On the other hand, the presence of forced reinsert operations increases the storage utilization and the search performance.

Few work has been undertaken on handling 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

1. For the hB-tree, it has been shown in [LS90] that under certain assumptions even a 33:67 splitting proportion yields an average storage utilization of 64%.

```
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 7: Algorithm for Exact Match Queries.

- Deleting the page and reinserting all objects into the index.

For most index structures it is a difficult task to find a suitable mate 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, Fal88, FR 89, Jag 90a] (c.f. Section 6.3 and Section 6.7). 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.

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 main memory. For all regions containing point q the function ExactMatchQuery() is called recursively. As overlap between page regions is allowed in most index structures presented in this chapter, it is possible that several branches of the indexing structure have to be examined for processing an exact match query. In the GiST framework [HNP 95], this situation is handled using the *Consistent* operation which is the generic operation that needs to be reimplemented for different instantiations of the generalized search tree. 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 7 contains the pseudocode for processing exact match queries.

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 using a weighted L_{max} metric.

The algorithm, presented in Figure 8, performs a recursive self-call for all child-pages, where the corresponding page regions intersect with the query. The union of the results of all recursive calls is built and passed to the caller.

3.4 Nearest Neighbor Query

There are two different approaches to processing of nearest neighbor queries on multidimensional index structures. One was published by Roussopoulos, Kelley and Vincent

```
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 8: Algorithm for Range Queries.

[RKV 95] and is in the following referred to as the RKV algorithm. The other, called HS algorithm, was published by Hjaltason and Samet [HS 95][1]. Due to their importance for our further presentation, 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 indexing structure 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 closeness to the query point.

Unlike range query processing, there is no fixed criterion, known *a priori*, to exclude branches of the indexing structure from processing in nearest neighbor algorithms. Actually, the criterion is the nearest neighbor distance but the nearest neighbor distance is not known until the algorithm has terminated. To cut branches, nearest neighbor algorithms have to use pessimistic (conservative) estimations of the nearest neighbor distance, which will change during the run of the algorithm and will approach to 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.e., minimum bounding rectangles, *MBR*) I with

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

Then, MINDIST is defined as follows:

1. A similar algorithm has been proposed by Henrich in [Hen 94].

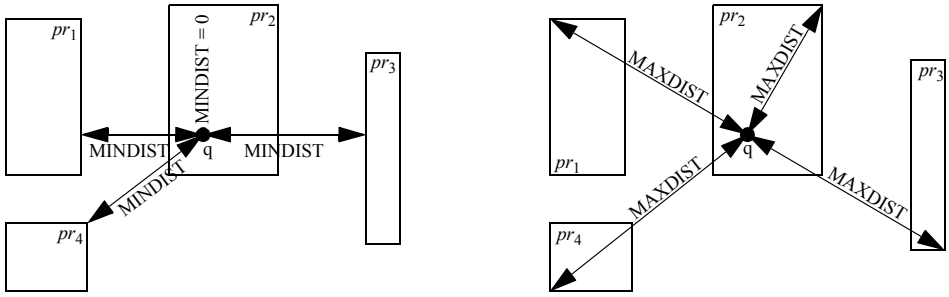


Figure 9: MINDIST and MAXDIST.

Definition 5 MINDIST.

The distance of a point q to region I, denoted 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 9. 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:

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

MAXDIST is not defined in the original paper, as it is not needed in R-tree like structures. An example is shown on the right side of Figure 9. Being the greatest possible distance from the query point to a point in a page region, the MAXDIST is not equal to 0, even if the query point is located inside the page region pr_2.

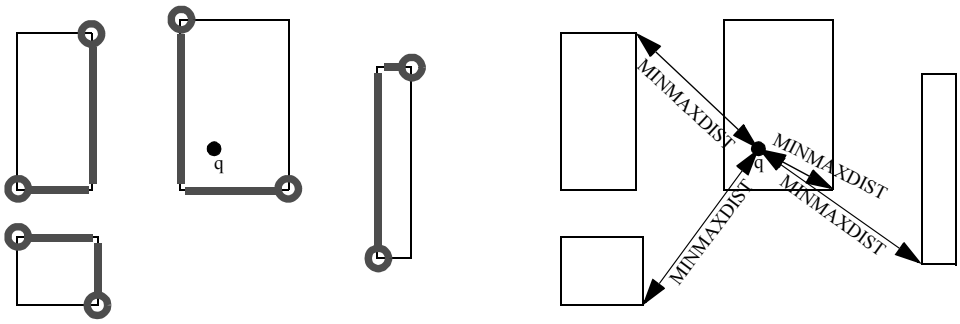


Figure 10: MINMAXDIST.

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

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

where:

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

The general idea is that every surface hyperarea must contain a point. The farthest point on every surface is determined and among those the minimum is taken. For each pair of opposite surfaces, only the nearer surface can contain the minimum. Thus, it is guaranteed that a data object can be found in the region having a distance less than or equal to MINMAXDIST (*q, I*). MINMAXDIST (*q, I*) is the smallest distance providing this guarantee. The example on Figure 10 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 10.

This pessimistic estimation cannot be used for spherical or combined regions as these in general do not fulfill a property similar to the MBR property. In this case, MAXDIST (*q, I*),

```
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 11: The RKV algorithm for finding the nearest neighbor.

which is an estimation worse than MINMAXDIST, has to be used. All definitions presented using 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 11 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 is begun. 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. These pages are guaranteed not to 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.

In [CF 98], Cheung and Fu discuss several heuristics for the RKV algorithm with and without the MINMAXDIST function. They prove that any page which can be pruned by exploiting the MINMAXDIST can also be pruned without that concept. Their conclusion is that the determination of MINMAXDIST should be avoided as it causes an additional overhead for the computation of MINMAXDIST.

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, as the algorithm has to assure, that *k* points are closer to the query than a given region. 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

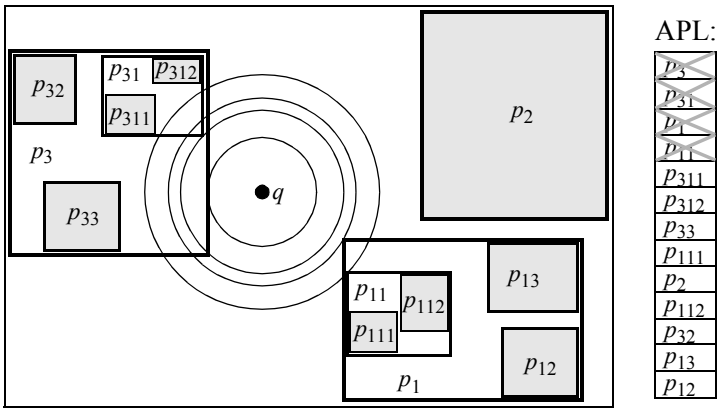


Figure 12: The HS algorithm for finding the nearest neighbor.

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 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 main memory.
- Delete p from the APL.
- If p is a data page, determine, if 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 *csf* due to the lower bounding property. Also, 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 important system limitation. We assume that the stack for recursion management and all priority queues are held in 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 see Appendix.

As the RKV algorithm performs a depth-first pass through the indexing 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 see Appendix.

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 main memory, although theoretically unattractive.

For the objective of 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 using counterexamples, 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 see Appendix.

Lemma 4. Schedule of the HS algorithm.

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

Proof see Appendix.

Lemma 5. Optimality of HS algorithm.

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

Proof see Appendix.

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 be begun. In the example of Figure 13, both algorithms chose *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.

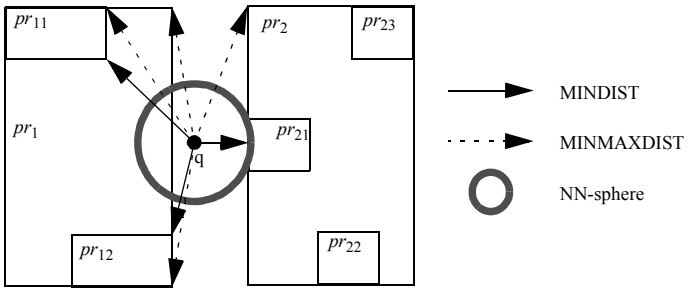


Figure 13: Schedules of RKV and HS Algorithm.

We have to summarize that the HS algorithm for nearest neighbor search is superior to the RKV algorithm when counting 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 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.

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 according to a criterion which is external to the index-based query processing. Therefore, neither a limited query range nor a limited result set size can be assumed before the application terminates the ranking query.

In contrast to the k-nearest neighbor algorithm, a ranking query algorithm needs an unlimited priority queue for the candidate list of closest points (*cpcl*). A further difference is that each request of the next closest point is regarded as a phase that is ended 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 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.

3.6 Reverse Nearest Neighbor Queries

In [KM 00], Korn and Muthukrishnan introduce the operation of *Reverse Nearest Neighbor Queries*. Given an arbitrary query point q, this operation retrieves all points of the database to which q is the nearest neighbor, i.e. the set of *reverse nearest neighbors*. Note that the nearest-neighbor relation is not symmetric: If some point p_1 is the nearest neighbor of p_2, then p_2 is not necessarily the nearest neighbor of p_1. Therefore, the result set of the rnn-operation can be empty or may contain an arbitrary number of points.

A database point p is in the result set of the rnn-operation for query point q unless another database point p' is closer to p than q is. Therefore, p is in the result set if q is enclosed by the sphere centered by p touching the nearest neighbor of p (the nearest neighbor sphere of p). Therefore, in [KM 00] the problem is solved by a specialized index structure for sphere objects which stores the nearest neighbor spheres rather than the database points. A rnn-query corresponds to a point query in that index structure. For an insert operation, the set of reverse nearest neighbors of the new point must be determined. The corresponding nearest neighbor spheres of all result points must be reinserted into the index.

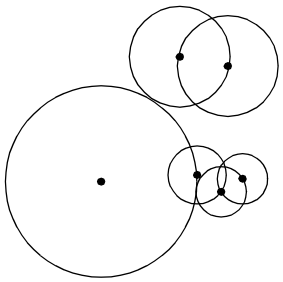


Figure 14: Indexing for the Reverse Nearest Neighbor Search.

The two most important drawbacks of this solution are the high cost for the insert operation and the use of a highly specialized index. For instance, if the rnn has to be determined only for a subset of the dimensions, a completely new index must be constructed. Therefore, Stanoi, Agrawal and Abbadi proposed in [SAA 00] a solution for point index structures. This solution, however, is limited to the 2-dimensional case.

4. Cost Models for High-Dimensional Index Structures

Due to the high practical relevance of multidimensional indexing, 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 data page accesses in an R-tree is

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

This formula is motivated as follows: The query evaluation algorithm is assumed to access an area of the data space which is a hyper cube of the volume $V_1 = 1/N$, where N is the number of objects stored in the database. Analogously, the page region is approximated by a hypercube with the volume $V_2 = C_{\text{eff}}/N$. In each dimension, the chance that the projection of V_1 and V_2 intersect each other, corresponds to $\sqrt[d]{V_1} + \sqrt[d]{V_2}$ if $n \to \infty$. To obtain a probability that V_1 and V_2 intersect in all dimensions, this term must be taken to the power of d. Multiplying this result with the number of data pages N / C_{eff}, yields the expected number of page accesses $A_{\text{nn,mm,FBF}}$. 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 approach 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 estimations. 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 factorial operator $x! = \Gamma(x+1)$ into the domain of real numbers: $\Gamma(x+1) = x \cdot \Gamma(x)$, $\Gamma(1) = 1$ and $\Gamma(\frac{1}{2}) = \sqrt{\pi}$.

For example, in a 20-dimensional data space with $C_{\text{eff}} = 20$, Sproull's formula evaluates to $N >> 1.1 \cdot 10^{11}$. We will see later (cf. figure 15), how bad the cost estimations of the FBF model are if substantially fewer than a hundred billion points are stored in the database. Unfortunately, Sproull still assumes for his analysis uniformity and independence in the distribution of data points and queries, i.e. both data points and the center points of the queries are chosen from a uniform data distribution, whereas the selectivity of the queries ($1/N$) is considered fix. The above formulas are also generalized to k-nearest neighbor queries, where k is also a user-given parameter.

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. 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 estimations. Figure 15 shows the actual page accesses for uniformly distributed point data versus the estimations of the model of Friedman, Bentley and Finkel. For high-dimensional data, the model completely fails to estimate the number of page accesses.

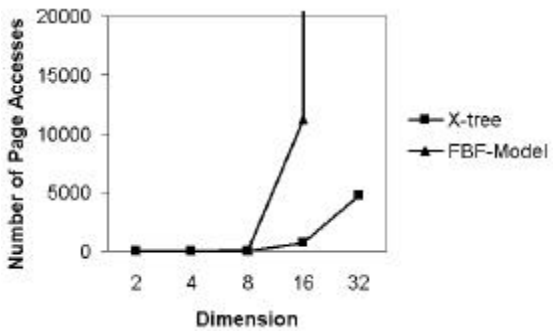


Figure 15: Evaluation of the model of Friedman, Bentley and Finkel.

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.

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 97a]. 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. 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. Note that in a 20-dimensional data space,

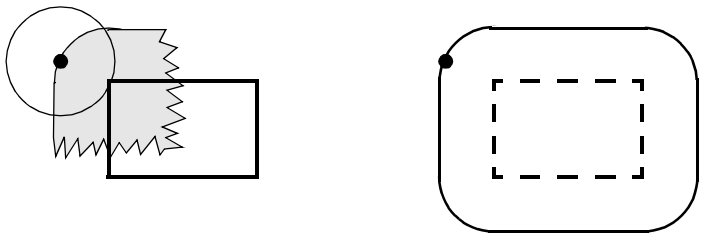


Figure 16: The Minkowski Sum.

the volume of the minimum bounding cube of a sphere is by a factor of $1 / V_S(1/2) = 4{,}1 \cdot 10^7$ larger than the volume of the sphere. The sphere volume, in turn, is by $V_S(\sqrt{d}/2) = 27,000$ times larger than the greatest enclosed cube. An 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.

Berchtold, Böhm, Keim and Kriegel [BBKK 97] presented in 1997 a cost model for query processing in high-dimensional data spaces, in the following called *BBKK model*. It provides accurate estimations for nearest neighbor queries and range queries using the Euclidean metric and considers boundary effects. 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 estimation 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. Ciaccia, Patella and Zezula [CPZ 98] adapt the cost model [BBKK 97] to estimate the page accesses of the M-tree, an index structure for data spaces which are metric spaces but not vector spaces (i.e. only the distances between the objects are known, but no explicit positions). Papadopoulos and Manolopoulos [PM 98] apply the cost model for declustering of data in a disk array. Two papers [RGF 98, AGGR 98] present applications in the data mining domain.

A recent paper [Boe 00] is based on the BBKK cost model which is presented in a comprehensive way and extended in many aspects. The extensions not yet covered by the BBKK model include all estimations for the maximum metric, which are developed additionally throughout the whole paper. The restriction of the BBKK model to numbers of data pages which are a power of two is overcome. A further extension of the model regards *k*-nearest neighbor queries (The BBKK model is restricted to 1-nearest-neighbor queries). The numerical methods for integral approximation and for the estimation of the boundary effects were to the largest extent out of the scope of [BBKK 97]. Finally, the concept of the fractal dimension, which was also used in the BBKK model in a simplified way (the data space dimension is simply replaced by the fractal dimension) is in this paper well established by the consequent application of the fractal power laws.

5. Indexing in Metric Spaces

In some applications, objects cannot be mapped into feature vectors. However, there still exists some notion of similarity between objects, which can be expressed as a metric distance between the objects, i.e. the objects are embedded in a metric space. The object distances can directly be used for query evaluation.

Several index structures for pure metric spaces have been proposed in the literature. Probably the oldest reference are the so-called Burkhard-Keller trees [BK 73]. Burkhard-Keller trees use a distance function which returns a small number *(i)* of discrete values. An arbitrary object is chosen as root of the tree and the distance function is used to partition the remaining data objects into *i* subsets which are the *i* branches of the tree. The same procedure is repeated for each non-empty subset to build up the tree (see Figure 17). More recently, a number of variants of the Burkhard-Keller tree have been proposed [BCMW 94]. In the fixed queries tree, for example, the data objects used as pivots are confined to be the same on the same level of the tree [BCMW 94].

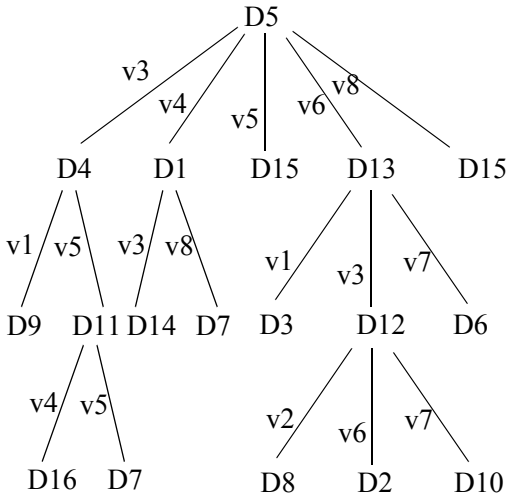


Figure 17: Example Burkhard-Keller-Tree
(D: data points, v: values of discrete distance function).

In most applications, a continuous distance function is used. Examples of index structures which are based on a continuous distance function are the vantage-point tree (VPT), the generalized hyperplane tree (GHT), and the M-tree. The VPT [Uhl 91, Yia 93] is a binary tree which uses some pivot element as the root and partitions the remaining data elements based on their distance with respect to the pivot element in two subsets. The same is repeated recursively for the subsets (see Figure 18). Variants of the VPT are the optimized VP-tree [Chi 94], the Multiple VP-tree [BO 97], and the VP Forest [Yia 99]).

The GHT [Uhl 91] is also a binary tree which uses two pivot elements on each level of the tree. All data elements which are closer to the first pivot element are assigned to the left subtree and all elements which are closer to the other pivot element are assigned to the other subtree (see Figure 19). A variant of the GHT is the geometric near-neighbor access tree

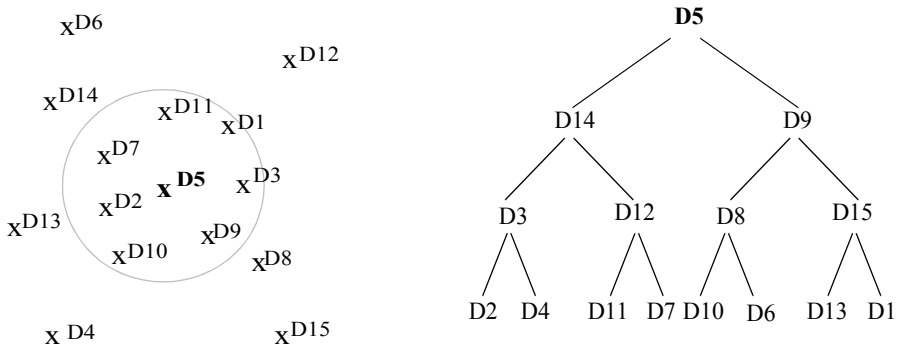


Figure 18: Example Vantage-Point Tree.

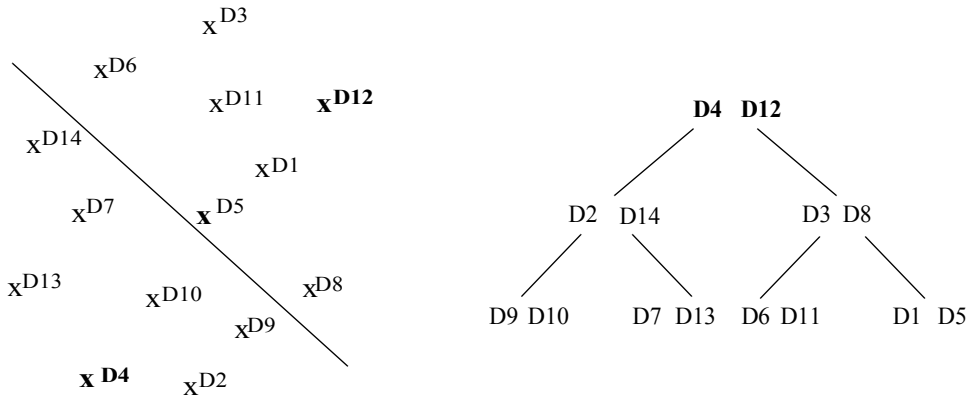


Figure 19: Example Generalized Hyperplane Tree.

(GNAT) [Bri 95]. The main difference is that the GNAT is an m-ary tree which uses m pivots on each level of the tree.

The basic structure of the M-tree [CPZ 97] is similar to the VP-tree. The main difference is that the M-tree is designed for secondary memory and allows overlap in the covered areas to allow easier updates. Note that among all metric index structures the M-tree is the only one which is optimized for large secondary-memory based data sets. All others are main memory index structures supporting rather small data sets.

Note that metric indexes are only used in applications where there the distance in vector space is not meaningful. This is true since vector spaces contain more information and therefore allows a better structuring of the data than general metric spaces.

6. Approaches to High-Dimensional Indexing

In this section, we will introduce and briefly discuss the most important index structures for high-dimensional data spaces. We will describe first 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 hashing-based methods [KS 86, KS 87, KS 88, Oto 84, NHS 84, Hin 85, KW 85, Ouk 85, Fre 87, HSW 88a, HSW 88b, HSW 89], are excluded from the discussion here. In the VAMSplit R-tree [JW 96] and in the Hilbert-R-tree [KF 94], methods for statically constructing R-trees are presented. Since the VAMSplit R-

tree and the Hilbert-R-tree is rather a construction method than an indexing structure of its own, they are also not presented in detail here.

6.1 R-tree, R*-tree, and R+-tree

The R-tree [Gut 84] family of index structures 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 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 into 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 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. 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. Details are provided in [GG 98]:

- An exponential algorithm,
- a quadratic algorithm,
- a linear algorithm, and
- Greene's algorithm [Gre 89].

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.

In the insert-algorithm, the suitable data page for the object is found in O (log n) time, by examining a single path of the index. At the one hand, it seems to be an advantage that only a single path is examined for the determination of the data page into which a point is inserted. An uncontrolled number of paths, in contrast, would violate the demand of an O(n log n) time complexity for the index construction. Figure 20 shows, however, that inserts are often

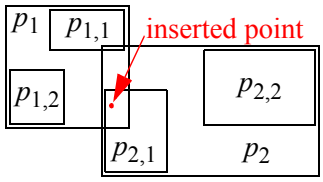


Figure 20: Misled Insert Operations.

misled in such tie situations. It is intuitively clear that the point must be inserted into page $p_{2,1}$, because $p_{2,1}$ is the only page on the second index level which contains the point. But the insert algorithm faces a tie situation at the first index level because both pages, p_1 as well as p_2 cover the point. According to the heuristics, the smaller page p_1 is chosen. The page $p_{2,1}$ as a child of p_2 will never be under consideration. The result of this misled insert is that the page $p_{1,2}$ unnecessarily becomes enlarged by a large factor and an additional overlap situation of the pages $p_{1,2}$ and $p_{2,1}$. Therefore, overlap at or near by the data level is mostly a consequence of some initial overlap in the directory levels near by the root (which would, eventually, be tolerable).

The initial overlap usually stems from the inability to split a higher-level page without overlap, because all child pages have independently grown extended page regions. For an overlap-free split, a dimension is needed in which the projections of the page regions have no overlap at some point. It has been shown in [BKK 96] that the existence of such a point becomes less likely as the dimension of the data space increases. The reason simply is that the projection of each child page to an arbitrary dimension is not much smaller than the corresponding projection of the child page. If we assume all page regions to be hypercubes of side length A (parent page) and a (child page), respectively we get $a = A \cdot \sqrt[d]{1/C_{\text{eff}}}$, which is substantially below A if d is small but actually in the same order of magnitude as A if d is sufficiently high.

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 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 form 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 index construction can be corrected this way.

Performance studies report improvements between 10 % and 75 % over the R-tree. In higher-dimensional data spaces, the split algorithm proposed in [BKSS 90] leads to a deteriorated directory. Therefore, the R*-tree is not adequate for these data spaces, rather it has to load the entire index in order to process most queries. A detailed explanation of this effect is given in [BKK 96]. The basic problem of the R-tree, overlap coming up at high index levels and then propagating down by misled insert operations is alleviated by more appropriate heuristics but not solved.

The heuristic of the R*-tree split to optimize for page regions with a small surface (i.e. for square/cube like page regions) is beneficial, in particular with respect to range queries and nearest neighbor queries. As pointed out in section 4 (cost models), the access probability corresponds to the Minkowski sum of the page region and the query sphere. The Minkowski sum primarily consists of the page region which is enlarged at each surface segment. If the page regions are optimized for a small surface, they directly optimize the Minkowski sum. Figure 21 shows an extreme nonetheless typical example of volume-equivalent pages and their Minkowski sums. The square (1×1 unit) yields with 3.78 a substantially lower Minkowski sum than the volume equivalent rectangle ($3 \times \frac{1}{3}$) with 5.11 units. Note again, that the effect becomes stronger with an increasing number of dimensions as every dimension is a potential source of unbalance. For spherical queries, however, spherical page regions yield the lowest Minkowski sum (3.55 units). Spherical page regions will be discussed later.

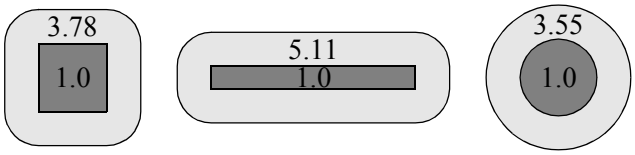


Figure 21: Shapes of Page Regions and their Suitability for Similarity Queries.

The R^+-tree [SSH 86; SRF 87] is an overlap-free variant of the R-tree. To guarantee no overlap the split algorithm is modified by a *forced-split* strategy. Child pages which are an obstacle in overlap-free splitting of some page, are simply cut into two pieces at a suitable position. It is possible, however, that these forced splits must be propagated down until the data page level is reached. The number of pages can even exponentially increase from level

to level. As we have pointed out before the extension of the child pages is not much smaller than the extension of the parent if the dimension is sufficiently high. Therefore, high dimensionality leads to many forced split operations. Pages which are subject to a forced split, are split although no overflow has occurred. The resulting pages are utilized by less than 50%. The more forced splits are raised, the more the storage utilization of the complete index will deteriorate.

A further problem which more or less concerns all of the data organizing techniques described in this paper is the decreasing fanout of the directory nodes with increasing dimension. For the R-tree family, for example, the internal nodes have to store $2d$ high and low bounds in order to describe a minimum bounding rectangle in d-dimensional space.

6.2 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 R*-trees for high-dimensional data. The major problem of R-tree-based index structures in high-dimensional data spaces is 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 case, a split should be avoided in order to avoid underfilled nodes.

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

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

If one records the history of data page splits in an R-tree based index structure, this results in a binary tree: The index starts with a single data page A covering almost the whole data space and inserts data items. If the page overflows, the index splits the page into two new pages A' and B. Later on, each of these pages might be split again into new pages. Thus, the history of all splits may be described as a binary tree, having split dimensions (and positions)

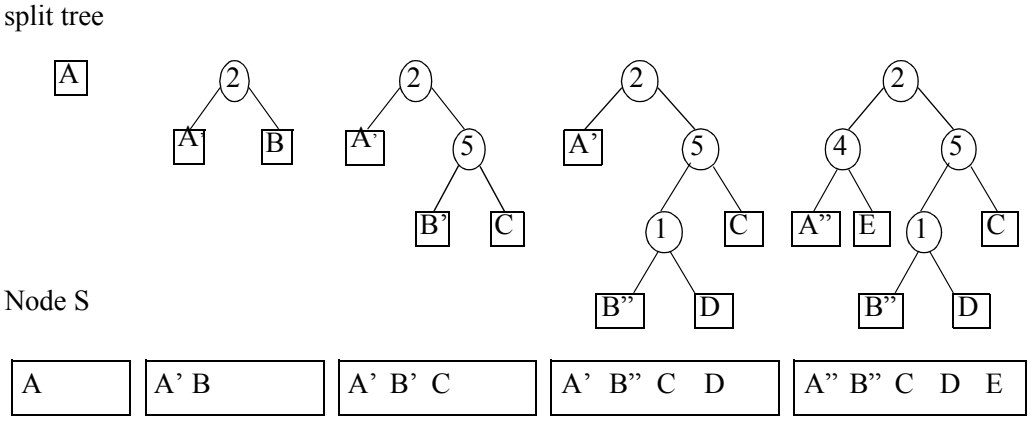


Figure 22: Example for the Split History.

as nodes and having the current data pages as leaf nodes. Figure 22 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 chose a split axis, first. Now, what are potential candidates for split axis in our example? Say, we chose 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. This obviously leads to high overlap with the other partition, regardless of the shape of the other partition. If one looks at the example in Figure 22, it becomes clear that only dimension 2 may be used as a split dimension. The X-tree generalizes this observation and uses always the split dimension with which the root node of the particular split tree is labeled. This guarantees an overlap free directory. However, the split tree might be unbalanced. In this case it is advantageous not to split at all because splitting would create one underfilled node and another almost overflowing node. Thus, the storage utilization in the directory would decrease dramatically and the directory would degenerate. In this case the X-tree does not split and creates an enlarged directory node instead, a supernode. The higher the dimensionality, the more supernodes will be created and the larger the supernodes become. To 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

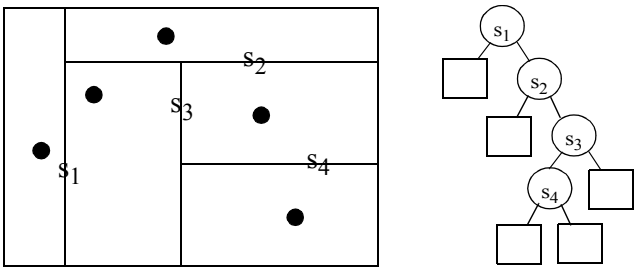


Figure 23: The kd-tree.

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 R*-trees for all query types in medium-dimensional spaces. For small dimensions, the X-Tree shows a behavior almost identical to 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 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.

6.3 Structures with a kd-tree directory

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 (c.f. Figure 23). 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 to use is always unambiguous. The deletion operation is also supported in a better way than in R-tree variants because 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, disjoint space partitioning.

Complete partitioning has the disadvantage that page regions are generally larger than necessary. Particularly in high-dimensional data spaces often large parts of the data space are not occupied by data points at all. Real data often are clustered or correlated. If the data distribution is cluster shaped, it is intuitively clear that large parts of the space are empty. But also the presence of correlations (i.e. one dimension is more or less dependent on the values of one or more other dimension) leads to empty parts of the data space, as depicted in

Figure 24: Incomplete vs. Complete Decomposition for Clustered and Correlated Data.

figure 24. Index structures which do not use complete partitioning are superior because larger page regions yield a higher access probability. Therefore, these pages are more often accessed during query processing than minimum bounding page regions. The second problem is, that kd-trees in principle 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 also be intersected by the split plane, which must be split either (*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.

Th hB-tree (*holey brick*) [LS 89; LS 90; Eva 94] also uses a kd-tree directory to define the page regions of the index. In this approach, splitting of a node is based on multiple attributes. This means that page regions do not correspond to solid rectangles but to rectangles from which other rectangles have been removed (*holey bricks*). With this technique, the forced split of the k-d-B-tree and the R^+-tree is avoided.

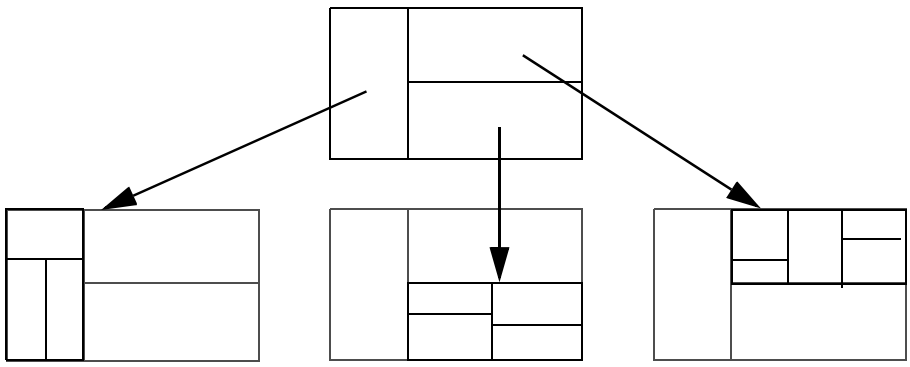


Figure 25: The k-d-B-tree.

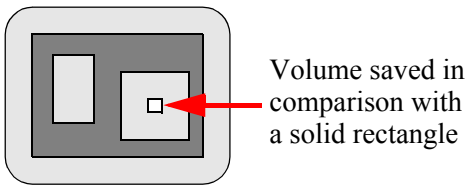


Figure 26: The Minkowski sum of a holey brick.

For similarity search in high-dimensional spaces, we can state the same benefits and shortcomings of a complete space decomposition as in the k-d-B-tree, depicted in figure 24. Additionally, we can state that the cavities of the page regions decrease the volume of the page region, but hardly decrease the Minkowski sum (and thus the access probability of a page). This is illustrated in figure 26, where two large cavities are removed from a rectangle, reducing its volume by more than 30%. The Minkowski sum, however, is not reduced in the left cavity, because it is not as wide as the perimeter of the query is. In the second cavity, there is only a very small area where the page region is not touched. Thus the cavities reduce the access probability of the page by less than 1%.

The directory of the LSD^h-tree [Hen 98] 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 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

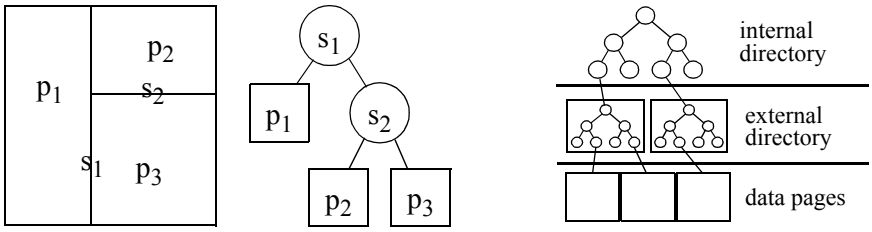


Figure 27: The LSD^h-tree.

description. While the space requirement of R-tree directory grows linearly with increasing dimension, it is constant (theoretically logarithmic, for very large dimensionality) for the LSD^h-tree. Note that this argument also holds for the hBpi-tree. See [ELS97] for a more detailed discussion of the issue. 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 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 using a value $z = 5$.

The most important advantage of the complete partitioning using potential data regions is that they allow a maintenance guaranteeing no overlap. It has been pointed out in the discussion of the R-tree variants and of the X-tree that overlap is a particular problem in high-dimensional data spaces. By the complete partitioning of the *kd*-tree directory, tie situations which lead to overlap do not arise. On the other hand, the regions of the index pages are not able to adopt equally well to changes in the actual data distribution as page regions can which are not forced into the *kd*-tree directory. The description of the 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 overlap between data pages in some cases. A situation where

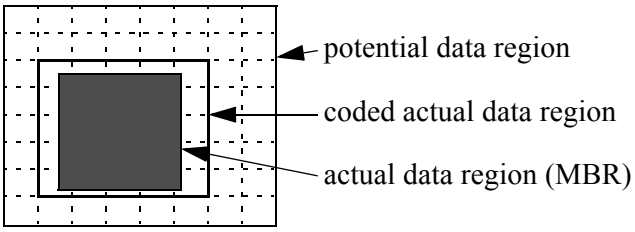


Figure 28: Region approximation using the LSD^h-tree.

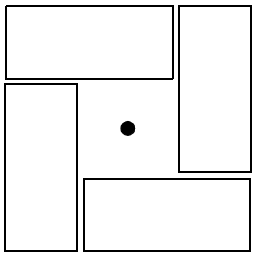


Figure 29: No overlap-free insert is possible.

no overlap-free enlargement is possible, is depicted in Figure 29. The coded actual data regions may have to be enlarged during an insert operation. As they are completely contained in a potential page regions, overlap cannot arise either.

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

As reported in [Hen 98], the LSD^h-tree show 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 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.

6.4 SS-tree

In contrast to all previously introduced index structures, the SS-tree [WJ 96] uses spheres as page regions. For maintenance 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. The region description comprises therefore the centroid point and the radius. This allows an efficient determination of the MINDIST and of 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*

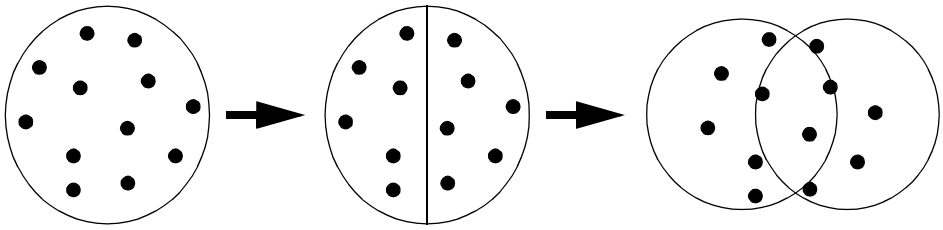


Figure 30: No overlap-free split is possible.

reinsert operation is raised, like in the R*-tree. 30% of the objects with 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 space utilization guarantees. The sum of the variances on each side of the split plane is minimized.

It was pointed out already in section 6.1 (cf. figure 21 in particular) that spheres are theoretically superior to volume-equivalent MBRs because the Minkowski sum is smaller. The general problem of spheres is, that they are not amenable to an easy, overlap-free split, as depicted in Figure 30. MBRs have in general a smaller volume, and, therefore, the advantage in the Minkowski sum is more than compensated. The SS-tree outperforms the R*-tree by a factor of 2, however, does not reach the performance of the LSD^h-tree and the X-tree.

6.5 TV-tree

The TV-tree [LJF 95] is designed especially for real data that are subject to the Karhunen-Loeve-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.

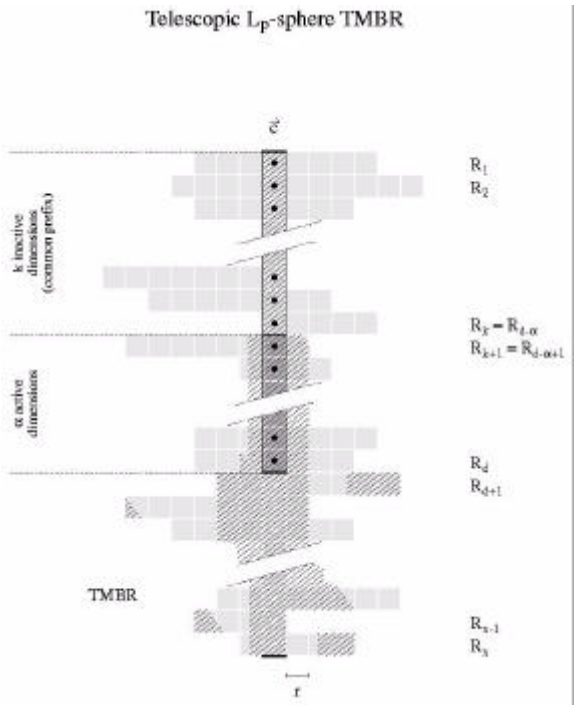


Figure 31: Telescope vectors.

Regions of the TV-tree are described using 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 31 depicts the extension of a telescope vector in space.

The region description comprises α floating point values for the coordinates of the center point in the active dimensions and one float value for the radius. The coordinates of the inactive dimensions are stored in higher levels of the index (exactly in the level, where a dimension turns from active into inactive). To achieve a uniform capacity of directory nodes, the number α of active dimensions is constant in all pages. The concept of telescope vectors increases the capacity of the directory pages. It was experimentally 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) having the least common prefix or (in case of doubt) having maximum distance. The objects are then inserted into one of the new subtrees using the above criteria for the subtree choice in insert processing, while 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).

6.6 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 point of the stored objects. Figure 32 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.

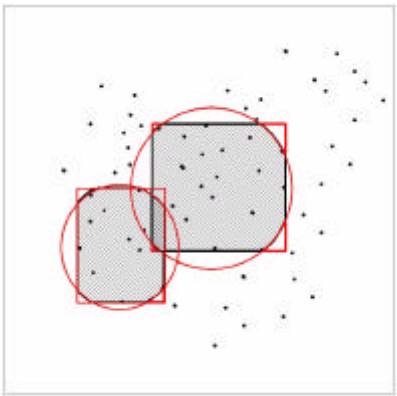


Figure 32: Page Regions of an SR-tree.

The motivation for using a combination of sphere and rectangle, presented by the authors is that according to an analysis presented in [WJ 96], spheres are basically better suited for processing nearest neighbor queries and range queries using the L_2-metric. On the other hand, spheres are difficult to maintain and tend to produce much overlap in splitting, as depicted previously in Figure 30. 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 33: Both distances to MBR and 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 $\text{MINDIST}(q, R)$ 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 MAXMINDIST definition exists for spheres, the MAXMINDIST 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

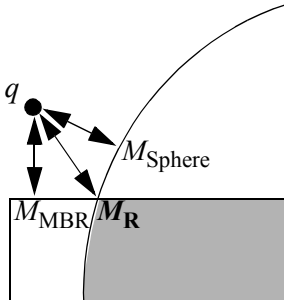


Figure 33: Incorrect MINDIST in the SR-tree.

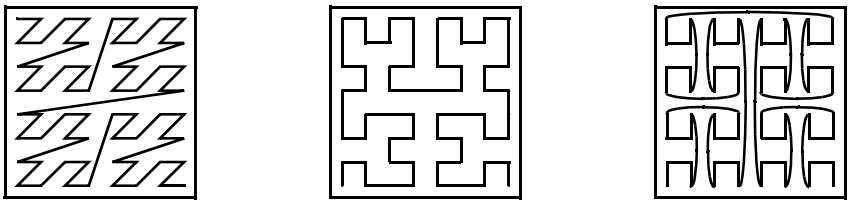
have to be updated and determined after inserts and node splits. Information of the MBRs is neither considered in the choice of branches nor in the determination of the split.

The reported performance results, compared to the SS-tree and the R*-tree, suggest that the SR-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.

6.7 Space Filling Curves

Space filling curves (for an overview see [Sag 94]) like Z-Ordering [Mor 66, FB 74, AS 83, OM 84, Ore 90], Gray Codes [Fal 85, Fal 88] or the Hilbert Curve [FR 89, Jag 90a, KF 93] 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 numbered. If only grid cells with the basic resolution *br* are considered, all bit strings have the same 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



Z-Ordering Hilbert Curve Gray Codes

Figure 34: Examples of Space Filling Curves.

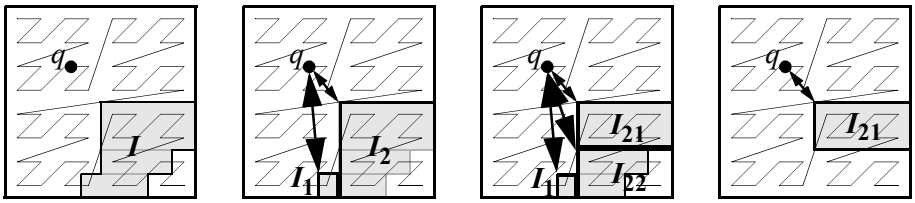


Figure 35: MINDIST determination using Space Filling Curves.

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 34.

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 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 using 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 $<p100...00>$. 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 $<p0>$ and $<p1>$. 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 35. 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, as 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.

6.8 Pyramid-tree

The Pyramid-Tree [BBK 98a] is an index structure that, similar to the Hilbert-technique, maps a d-dimensional point into a 1-dimensional space and uses a B+-tree to index the 1-dimensional space. Obviously, queries have to be translated in the same way. In the data pages of the B+-tree, the Pyramid-tree stores both, the d-dimensional points and the 1-dimensional key. Thus, no inverse transformation is required and the refinement step can be done without look-ups to another file. The specific mapping used by the Pyramid-tree is called Pyramid-mapping. It is based on a special partitioning strategy that is optimized for range queries on 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 be efficiently stored by R-tree-like or kd-tree-like index structures. However, the Pyramid-tree achieves the partitioning by first dividing the *d*-dimensional space into 2*d* pyramids having the center point of the space as their top. In a second step, the single pyramids are cut into slices parallel

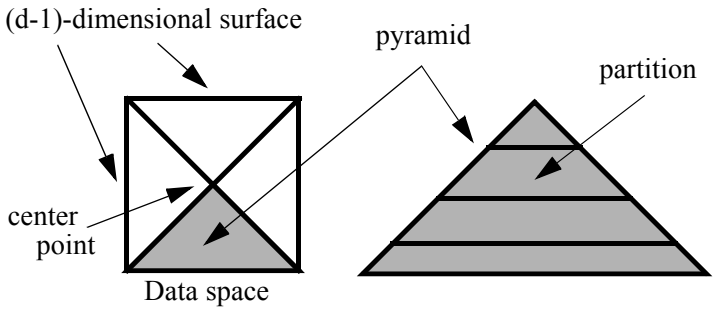
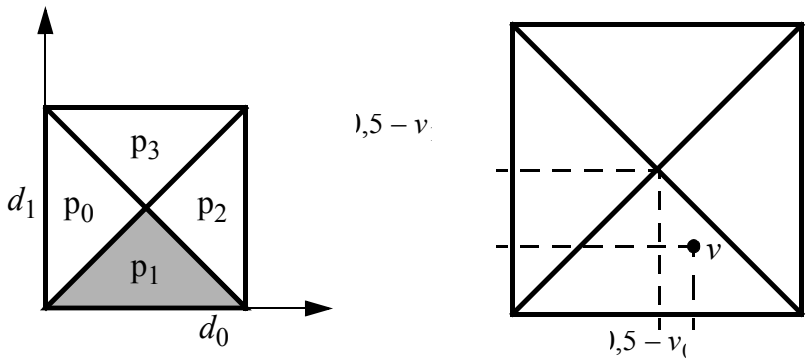


Figure 36: Partitioning the Data Space into Pyramids.



a) numbering of pyramids **b**) point in pyramid

Figure 37: Properties of Pyramids.

to the basis of the pyramid forming the data pages. Figure 36 depicts this partitioning technique.

This technique can be used to compute a mapping as follows: In the first step, we number the pyramids as shown in figure 37 a). Given a point, it is easy to determine in which pyramid is located. Then, we determine the so-called height of the point within its pyramid that is the orthogonal distance of the point to the center point of the data space (as shown in figure 37 b). In order to map a d-dimensional point into a 1-dimensional value, we simply add the two numbers, the number of the pyramid in which the point is located, and the height of the point within this pyramid.

Query processing is a non-trivial task on a Pyramid-tree because for a given query range, we have to determine the affected pyramids and the affected heights within these pyramids. The details of how this can be done are explained in [BBK 98a]. Although the details of the algorithm are hard to understand, it is not computationally hard, rather it consists of a variety of cases that have to be distinguished and simple computations.

The Pyramid-tree is the only index structure known so far that is not affected by the so-called curse of dimensionality. This means that for uniform data and range queries, the performance of the Pyramid-tree is getting even better if one increases the dimensionality of the data space. An analytical explanation of this phenomenon is given in [BBK 98a].

6.9 Summary

Table 1 shows the index structures described above and their most important properties. The first column contains the name of the index structure, the second shows which geometrical region is represented by a page, the third and fourth column show if the index structure

provides a disjoint and complete of the partitioning of the data space. The last three columns describe the used algorithms: What strategy is used to insert new data items (column 5), what criteria are used to determine the division of objects into sub-partitions in case of a overflow (column 6), and if the insert algorithm uses the concept of forced reinserts (column 7).

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
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
Pyramid-tree	trunks of pyramids	yes	yes	(unique due to complete, disjoint part.)	according to pyramid-mapping	no

Table 1: High-dimensional index structures and their properties

Since, so far, no extensive and objective comparison between the different index structures was published, only structural arguments may be used in order to compare the different approaches. The experimental comparisons tend to highly depend on the data that has been used in the experiments. Even higher is the influence of seemingly minor parameters such as the size and location of queries or the statistical distribution of these. The higher the dimensionality of the data, the more these influences lead to different results. Thus, we provide a comparison between the indexes listing only properties not trying to say anything about the "overall" performance of a single index. In fact, most probably, there is no overall performance, rather, one index will outperform other indexes in a special situation whereas this index is quite useless for other configurations of the database. Table 2 shows such a compar-

ison. The first column lists the name of the index, the second column explains the biggest problem of this index when the dimension increases. The third column lists the supported types of queries. In the fourth column, we show if a split in the directory causes "forced splits" on lower levels of the directory. The fifth column shows the storage utilization of the index, which is only a statistical value depending on the type of data and, sometimes, even on the order of insertion. The last column is about the fanout in the directory which in turn depends on the size of a single entry in an directory node.

Name	problems in high-D	supported query types	locality of node splits	storage utilization	fanout / size of index entries
R-tree	poor split algorithm leads to deteriorated directories	NN, region, range	yes	poor	poor, linearly dimension dependent
R*-tree	dto.	NN, region, range	yes	medium	poor, linearly dimension dependent
X-tree	high probability of queries overlapping MBR's leads to poor performance	NN, region, range	yes	medium	poor, linearly dimension dependent
LSD^h-tree	changing data distribution deteriorates directory	NN, region, range	no	medium	very good, dimension independent
SS-tree	high overlap in directory	NN	yes	medium	very good, dimension independent
TV-tree	only useful for specific data	NN	yes	medium	poor, somehow dimension dependent
SR-tree	very largedirectory sizes	NN	yes	medium	very poor, linearly dimension dependent
space filling curves	poor space partitioning	NN, region, range	yes	medium	as good as B-Tree, dimension independent
Pyramid-tree	problems with asymetric queries	region, range	yes	medium	as good as B-Tree, dimension independent

Table 2: Qualitative Comparison of High-dimensional index structures

7. Improvements, Optimizations, and Future Research Issues

During the past years, a significant amount of work has been invested not to develop new index structures but to improve the performance of existing index structures. As a result, a

variety of techniques have been proposed using or tuning index structures. In this section, we present a selection of those techniques. Furthermore, we will point out a selection of problems that have not yet been addressed in the context of high-dimensional indexing, or the solution of which can not be considered sufficient.

Tree-Striping

From a variety of cost models that have been developed one might conclude that if the data space has a sufficiently high dimensionality, no index structure can succeed. On the one hand, this has been contradicted by the development of index structures that are not severely affected by the dimensionality of the data space. On the other hand, one has to be very careful to judge the implications of a specific cost model: A lesson all researchers in the area of high-dimensional index structures learned was that things are very sensitive to the change of parameters. A model modeling nearest neighbor queries can not directly be used to make any claims about the behavior in case of range queries. Still, the research community agreed that in case of nearest neighbor queries, there exists a dimension above which a sequential scan will be faster than any indexing technique for most relevant data distributions.

Tree-Striping is a technique that tries to tackle the problem from a different perspective: If it is hard to solve the d-dimensional problem of query processing, why not try to solve k l-dimensional problems, where $k \cdot l = d$. The specific work presented in [BBK+ 00] focuses on the processing of range queries in a high-dimensional space. It generalizes the well-known inverted lists and multidimensional indexing approaches. A theoretical analysis of the generalized technique shows that both, inverted lists and multidimensional indexing approaches, are far from being optimal. A consequence of the analysis is that the use of a set of multidimensional indexes provides considerable improvements over one d-dimensional index (multidimensional indexing) or d one-dimensional indexes (inverted lists). The basic idea of tree striping is to use the optimal number k of lower-dimensional indexes determined by a theoretical analysis for efficient query processing. A given query is also split into k lower-dimensional queries and processed independently. In a first step, the single results are merged. As the merging step also involves I/O cost and these cost increase with a decreasing dimensionality of a single index, there exists an optimal dimensionality for the single indexes that can be determined analytically. Note that tree-striping has serious limitations especially for nearest-neighbor queries and skewed data where, in many cases, the d-dimensional index performs better than any lower-dimensional index.

Voronoi-Approximations

In another approach [BEK+ 98, BKKS 00] to overcome the curse of dimensionality for nearest neighbor search, the results of any nearest-neighbor search are pre-computed. This corresponds to a computation of the voronoi cell of each data point. The voronoi cell of a point *p* contains all points that have *p* as a nearest neighbor. In high-dimensional spaces, the exact computation of a voronoi cell is computationally very hard. Thus, rather than computing exact voronoi cells, the algorithm stores conservative approximations of the voronoi cells in an index structure that is efficient for high-dimensional data spaces. As a result, nearest neighbor search corresponds to a simple point query on the index structure. Although the technique is based on a pre-computation of the solution space, it is dynamic i.e. it supports insertions of new data points. Furthermore, an extension of the technique to a k-nearest neighbor search is given in [BKKS 00].

Parallel Nearest-Neighbor Search

Most similarity search techniques map the data objects into some high-dimensional feature space. The similarity search then corresponds to a nearest-neighbor search in the feature space which is computationally very intensive. In [BBB+ 97], the authors present a new parallel method for fast nearest-neighbor search in high-dimensional feature spaces. The core problem of designing a parallel nearest-neighbor algorithm is to find an adequate distribution of the data onto the disks. Unfortunately, the known declustering methods do not perform well for high-dimensional nearest-neighbor search. In contrast, the proposed method has been optimized based on the special properties of high-dimensional spaces and therefore provides a near-optimal distribution of the data items among the disks. The basic idea of this data declustering technique is to assign the buckets corresponding to different quadrants of the data space to different disks. The authors show that their technique - in contrast to other declustering methods - guarantees that all buckets corresponding to neighboring quadrants are assigned to different disks. The specific mapping of points to disks is done by the following formula:

$$\text{col}(c) = \left(\operatorname*{XOR}_{i=0}^{d-1} \left(\begin{cases} i+1 & \text{if } c_i = 1 \\ 0 & \text{otherwise} \end{cases} \right) \right)_{10}$$

The input is a bit string defining the quadrant in which the point to be declustered is located. But not any number of disks may be used for this declustering technique. In fact, the number required is linear in the number of the dimensions. Therefore, the authors present an extension of their technique adapted to an arbitrary number of disks. A further extension is

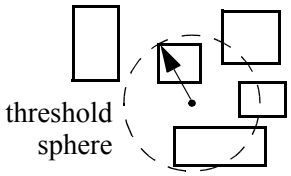


Figure 38: The threshold sphere for FPSS and CRSS.

a recursive declustering technique which allows an improved adaptation to skewed and correlated data distributions.

An approach for similarity query processing using disk arrays is presented in [PM 98]. The authors propose two new algorithms for the nearest neighbor search on single processors and multiple disks. Their solution relies on a well-known page distribution technique for low dimensional data spaces [KF 92] called *proximity index*. Upon a split, the MBR of a newly created node is compared with the MBRs which are stored in its father node (i.e. its siblings). The new node is assigned to the disk which stores the "least proximal" pages with respect to the new page region. Thus, the selected disk must contain sibling nodes which are far from the new node. The first algorithm, called *full parallel similarity search* (FPSS) determines the *threshold sphere* (cf. figure 38), an upper bound of the nearest neighbor distance according to the maximum distance between the query point and the nearest page region. Then, all pages which are not pruned by the threshold sphere are called in by a parallel request to all disks. The second algorithm, *candidate reduction similarity search* (CRSS), applies a heuristic which leads to an intermediate form between depth-first search and breadth-first search of the index. Pages which are completely contained in the threshold sphere are processed with a higher priority than pages which are merely intersected by it. The authors compare FPSS and CRSS with a (not existing) optimal parallel algorithm which knows the distance of the nearest neighbor in advance and report up to 100% more page accesses of CRSS compared to the optimal algorithm. The same authors also propose a solution for shared-nothing parallel architectures [PM 97b]. Their architecture distributes the data pages of the index over the secondary servers while the complete directory is held in the primary server. Their static page distribution strategy is based on a fractal curve (sorting according to the Hilbert value of the centroid of the MBR). The k-nn algorithm first performs a depth-first search in the directory. When the bottom level of the directory is reached, a sphere around the query point is determined which encloses as many data pages which are required to guarantee that k points are stored in them (i.e. assuming that the page capacity is $\geq k$, the sphere is chosen such that 1 page is completely contained). A parallel

range query is performed, first accessing a smaller number of data pages obtained by a cost model.

Compression Techniques

Recently, the VA file [WSB 98] was developed, an index structure that is actually not an index structure. Based on the cost model proposed in [BBKK 97] the authors prove that under certain assumptions, above a certain dimensionality no index structure can process a nearest neighbor query efficiently. Therefore, they suggest to accelerate the sequential scan by the use of data compression. The basic idea of the VA-file is to keep two files: A bit compressed, quantized version of the points and their exact representation. Both files are unsorted, however, the positions of the points in the two files agree.

The quantization of the points is determined by an irregular grid which is laid over the data space. The resolution of the grid in each dimension corresponds to 2^b, where b is the number of bits per dimension which are used to approximate the coordinates. The grid lines correspond to the quantiles of the projections of the points to the corresponding axes. These quantiles are assumed to change rarely. Changing the quantiles requires a reconstruction of the compressed file. The k-nearest neighbor queries are processed by the multi-step paradigm: The quantized points are loaded into main memory by a sequential scan (filter step). Candidates which cannot be pruned are refined, i.e. their exact coordinates are called in from the second file. Several access strategies for timing filter and refinement steps have been proposed. Basically, the speed-up of the VA-file compared to a simple sequential scan corresponds to the compression rate, because reading large files sequentially from disk yields a linear time complexity with respect to the file length. The computational effort of determining distances between the query point and the quantized data points is also improved compared to the sequential scan by precomputation of the squared distances between the query point and the grid lines. CPU speed-ups, however, do not yield large factors and are independent of the compression rate. The most important overhead in query processing are the refinements which require an expensive random disk access each. With decreasing resolution, the number of points to be refined increases, thus limiting the compression ratio. The authors report a number of 5-6 bits per dimension to be optimal.

There are two major drawbacks of the VA-file. First: The deterioration of index structures is much more prevalent in artificial data than in data sets from real-world applications. For such data, index structures are efficiently applicable for much higher dimensions. The second drawback is the number of bits per dimension which is a system parameter. Unfortunately, the authors do not provide any model or guideline for the selection of a suitable bit rate. To overcome these drawbacks, the IQ-tree [BBJ+ 00] has recently been proposed,

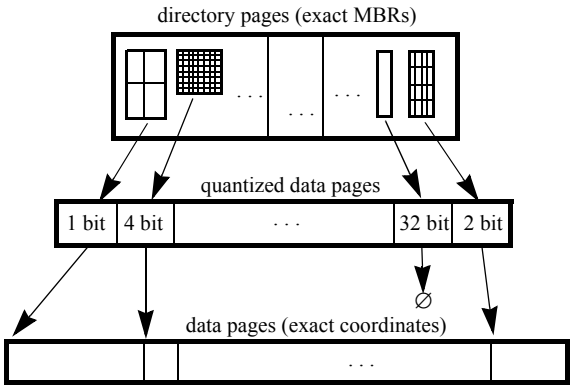


Figure 39: : Structure of the IQ-tree.

which is a 3-level tree index structure exploiting quantization (cf. figure 39). The first level is a flat directory consisting of MBRs and the corresponding pointers to pages on the second level. The pages on the second level contain the quantized version of the data points. In contrast to the VA-file, the quantization is not based on quantiles but is a regular decomposition of the page regions. The authors claim that regular quantization based on the page regions adopts equally well to skewed and correlated data distributions as quantiles do. The suitable compression rate is determined for each page independently according to a cost model which is proposed in [BBJ+ 00]. Finally, the bottom level of the IQ tree contains the exact representation of the data points. For processing of nearest neighbor queries, the authors propose a fast index scan which essentially subsumes the advantages of indexes and scan based methods. The algorithm collects accesses to neighboring pages and performs chained IO requests. The length of such chains is determined according to a cost model. In situations, where a sequential scan is clearly indicated, the algorithm degenerates automatically to the sequential scan. In other situations, where the search can be directly targeted, the algorithm performs the priority search of the HS algorithm. In intermediate situations, the algorithm accesses chains of intermediate length thus clearly outperforming both, the sequential scan as well as the HS algorithm. The bottom level of the IQ tree is accessed according to the usual multistep strategy.

Bottom-Up Construction

Usually, the performance of dynamically inserting a new data point into a multidimensional index structure is poor. The reason for this is that most structures have to consider multiple paths in the tree, where the point could be inserted. Furthermore, split algorithms are complex and computationally intensive algorithms. For example, a single split in an X-tree might take up to the order of a second to be performed. Therefore, a number of bulk-load algorithms for multidimensional index structures have been proposed. Bulk-loading an in-

dex means to build an index on an entire database in a single process. This can done much more efficiently than inserting the points one at a time. Most of the bulk-load algorithms such as the algorithm proposed in [BSW 97] are not especially adapted for the case of high-dimensional data spaces. In [BBK 98] however, Berchtold et. al. proposed a new bulk-loading technique for high-dimensional indexes that exploits a priori knowledge of the complete data set to improve both construction time and query performance. The algorithm operates in a manner similar to the Quicksort algorithm and has an average run time complexity of $O(n \log n)$. In contrast to other bulk-loading techniques, the query performance is additionally improved by optimizing the shape of the bounding boxes, by completely avoiding overlap, and by clustering the pages on disk. A sophisticated unbalanced split strategy is used leading to a better space partitioning.

Another important issue would be to apply the knowledge that people aggregated to other areas such as data reduction, data mining (e.g. clustering), or visualization, where people have to deal with tens to hundreds of attributes and therefore face a high-dimensional data space. Most of the lessons learned also apply to these areas. Examples for successful approaches to make use of these side-effects are [BJK 98] and [AGGR 98].

Future Research Issues

Although a significant progress has been made to understand the nature of high-dimensional spaces and to develop techniques that can operate in these spaces, there still are many open questions.

A first problem is that most of the understanding that the research community developed during the last years is restricted to the case of uniform and independent data. Not only are all proposed indexing techniques optimized for this case, also almost all theoretical considerations such as cost models are restricted to this simple case. The interesting observation is that index structures do not suffer from “real” data. Rather, they nicely take advantage from having non-uniform distributions. In fact, a uniform distribution seems to the worst thing that can happen to an index structure. One reason for this effect is that often the data is located only on a subspace of the data space and if the index adapts to this situation, it actually behaves as if the data would be lower-dimensional. A promising approach to understand and explain this effect theoretically has been followed in [FK 94, Boe 98] where the concept of the fractal dimension is applied. However, even this approach cannot cover “real” effects such as local skewness.

A second interesting research issues are the partitioning strategies that perform well in high-dimensional spaces. As previous research (e.g., the Pyramid-Tree) has shown, the partitioning does not have to be balanced to be optimal for certain queries. The open question

is what an optimal partitioning schema for nearest-neighbor queries would be? Does it need to be balanced or better unbalanced? Is it based upon bounding-boxes or on pyramids? How does the optimum change when the data set grows in size or dimensionality? There are many open question which need to be answered.

A third open research issue is an approximate processing of nearest-neighbor queries. The question is first of all what a useful definition for approximate nearest-neighbor search in high-dimensional spaces is, and how that fussiness introduced by the definition may be exploited for an efficient query processing. A first approach has for an approximate nearest-neighbor search been proposed in [GIM 99].

Other interesting research issues include the parallel processing of nearest-neighbor queries in high-dimensional space and the data mining and visualization of high-dimensional spaces. The parallel processing aims at finding appropriate declustering and query processing strategies to overcome the difficulties in high-dimensional spaces. A first approach in this direction has already been presented in [BBB+ 97]. The efforts in the area of data mining and visualization of high-dimensional feature spaces (for an example see e.g. [HK 98]) try to understand and explore the high-dimensional feature spaces. Also, the application of compression techniques to improve the query performance is an interesting and promising research area. A first approach, the VA-file, has recently been proposed in [WSB 98].

8. Conclusions

Research in high-dimensional index structures has been very active and productive over the past few years, resulting in a multitude of interesting new approaches for indexing high-dimensional data. Since it is very difficult to follow up on this discussion, in this survey we tried to provide insight in the effects occurring in indexing high-dimensional spaces and to provide an overview of the principle ideas of the index structures which have been proposed to overcome these problems. There are still a number of interesting open research problems and we expect the field to remain a fruitful research area over the next years. Due to the increasing importance of multimedia databases in various application areas and due to the remarkable results of the research, we also expect the research on high-dimensional indexing to have a major impact on many practical applications and commercial multimedia database systems.

9. Appendix

Lemma 1. Worst case space complexity of the RKV algorithm

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

Proof:

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, Lemma 1 follows. QED.

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:

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*). QED.

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:

(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. QED.

Lemma 4. Schedule of the HS algorithm.

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

Proof:

Due to the lower bounding property of page regions, the distance between the query point and a page region is always greater than or equal to the distance of the query point and the region of the parent of the page is. 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. QED.

Lemma 5. Optimality of HS algorithm.

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

Proof:

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 than 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. QED.

10. Literature

[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., Sawhney 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.

[Aok 98] Paul M. Aoki. Generalizing ``Search'' in Generalized Search Trees. Proc. 14th Int'l Conf. on Data Engineering, Orlando, FL, Feb. 1998, pp. 380-389.

[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*, Barcelona, Catalonia, 1991, 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*, SIGMOD best paper award, 1997.

[BBJ+ 00] Berchtold S., Böhm C., Jagadish H.V., Kriegel H.-P. and Sander J. Independent quantization: An index compression technique for high-dimensional data spaces. *Proc. 16th Int. Conf. on Data Engineering*, 2000.

[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 98a] 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+ 00] Berchtold S., Böhm C., Keim D., Kriegel H.-P., Xu X.:*'Optimal Multidimensional Query Processing Using Tree Striping',* submitted for publication.

[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.

[BBKK 00] Berchtold S., Böhm C., Keim D., Kriegel H.-P.: '*Optimized Processing of Nearest Neighbor Queries in High-Dimensional Data Space*', submitted for publication.

[BCMW 94] Baeza-Yates R. Cunto W., Manber U., Wu S.: *'Proximity Matching using fixed-queries trees'*, Proc. Combinatorial Pattern Matching (CPM'94), LNCS 807, pp. 198-212, 1994.

[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.: *'Multidimensiuonal Binary Search in Database Applications'*, IEEE Trans. Software Eng. 4(5), 1979, pp. 397-409.

[BF 95] Belussi A., Faloutsos C. Estimating the selectivity of spatial queries using the correlation fractal dimension. *Proc. 21st Int. Conf. on Very Large Data Bases*, Zurich, Switzerland, 1995, 299-310.

[BGRS 99] Beyer K., Goldstein J., Ramakrishnan R., Shaft U..: *'When Is "Nearest Neighbor" Meaningful?'*, Proc. Int. Conf. on Database Theory (ICDT), 1999, pp. 217-235.

[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 73] Burkhard W., Keller R.: *'Some approaches to best-match file searching'*, Comm. of the ACM, Vol. 16, No. 4, pp. 230-236, 1973.

[BK 98] Berchtold S., Keim D.: *'High-Dimensional Index Structures - Database Support for Next Decade's Applications'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Seattle, NJ, 1998.

[BK 00] Berchtold S., Keim D.: *'High-Dimensional Index Structures - Database Support for Next Decade's Applications'*, Proc. Int. Conf. on Data Engineering, San Diego, CA, 2000.

[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.

[BKKS 00] Berchtold S., Keim D., Kriegel H.-P., Seidl T.: '*Indexing the Solution Space: A New Technique for Nearest Neighbor Search in High-Dimensional Space',* Trans. on Knowledge and Data Engineering (TKDE), 2000.

[BKSS 90] Beckmann N., Kriegel H.-P., Schneider R., Seeger B.: *'The R*-tree: An Efficient and Robust Access Method for Points and Rectangles',* Proc. ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, NJ, 1990, pp. 322-331.

[BM 77] Bayer R., McCreight E.M.: *'Organization and Maintenance of Large Ordered Indices'*, Acta Informatica 1(3), 1977, pp. 173-189.

[BO 97] Bozkaya T., Ozsoyoglu M.: *'Distance-based indexing for high-dimensional metric spaces',* Proc. ACM SIGMOD International Conference on Management of Data, Sigmod Record Vol. 26, No. 2, pp. 357-368, 1997.

[Boe 98] Böhm C.: *'Efficiently Indexing High-Dimensional Data Spaces'*, Ph.D. thesis, University of Munich, 1998.

[Boe 00] Böhm C. *A Cost Model for Query Processing in High-Dimensional Data Spaces.* To appear in: ACM Transactions on Database Systems, 2000.

[Bri 95] Brin S.: '*Near Neighbor Search in Large Metric Spaces*', Proc. 21th Int. Conf. on Very Large Databases (VLDB), pp. 574-584, Switzerland, 1995.

[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.

[CF 98] Cheung K. L., Fu A. W.: '*Enhanced Nearest Neighbour Search on the R-tree'*, SIGMOD Record 27(3), 1998, pp. 16-21.

[Chi 94] Chiueh T.: '*Content-based image indexing*', Proc. 20th Int. Conf. on Very Large Databases (VLDB), pp. 582-593, Chile, 1994.

[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.

[CMTV 00] Corral A., Manolopoulos Y., Theodoridis Y., Vassilakopoulos M.: '*Closest Pair Queries in Spatial Databases*', ACM SIGMOD Int. Conf. on Management of Data, pp. 189-200, 2000.

[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. 23th Int. Conf. on Very Large Databases (VLDB), pp. 426-435, Greece, 1997.

[CPZ 98] Ciaccia P., Patella M., Zezula P.: '*A Cost Model For Similarity Queries in Metric Spaces*', Proc. 17th ACM Symopsium on Principles of Database Systems (PODS), pp. 59-67, Seattle, 1998.

[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.

[ELS 97] Evangelidis G., Lomet D. Salzberg B.: *'The hB^{pi}-Tree: A multiattribute Index Supporting Concurrency, Recovery and Node Consolidation'*, The VLDB Journal 6 (1), 1-25, 1997.

[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.

[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. *Proc. 22nd Int. Conf. on Very Large Data Bases*, Mumbai (Bombay), India, 1996, 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.

[Gae 95] Gaede V. Optimal redundancy in spatial database systems. *Proc. 4th International Symposium on Advances in Spatial Databases*, Portland, Maine, 1995, 96-116.

[GG 98] Gaede V., Günther O.: *'Multidimensional Access Methods*', ACM Computing Surveys, Vol. 30, No. 2, 1998, pp. 170-231.

[GIM 99] Gionis A., Indyk P., Motwani R.: 'Similarity Search in High Dimensions via Hashing', Proc. 25th Int. Conf. on Very Large Data Bases, Edinburgh, GB, pp. 518-529, 1999.

[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.

[HK 98] Hinneburg A., Keim D. A.: *'An Efficient Approach to Clustering in Large Multimedia Databases with Noise'*, Int. Conf. on Knowledge Discovery in Databases (KDD'98), New York, 1998.

[HKP 97] Joseph M. Hellerstein, Elias Koutsoupias and Christos H. Papadimitriou. On the Analysis of Indexing Schemes. Proc. 16th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, Tucson, AZ, May 1997, pp. 249-256.

[HNP 95] Joseph M. Hellerstein, Jeffrey F. Naughton and Avi Pfeffer. Generalized Search Trees for Database Systems. Proc. 21st Int'l Conf. on Very Large Data Bases, Zürich, September 1995, pp. 562-573.

[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.

[HS 98] Hjaltason G. R., Samet H.: '*Incremental Distance Join Algorithms for Spatial Databases*', *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 237-248, 1998.

[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, pp. 45-53, 1989.

[Jag 90a] 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.

[KF 92] Kamel I. and Faloutsos C. Parallel R-trees. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 195-204, 1992.

[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.

[KM 00] Korn F., Muthukrishnan S.: '*Influence Sets Based on Reverse Nearest Neighbor Queries',* ACM SIGMOD Int. Conf. on Management of Data, 2000, pp. 201-212.

[Knu 75] Knuth D.: *'The Art of Computer Programming - Volume 3: Sorting and Searching'*, Addison Wesley, 1975.

[Kor 99] Marcel Kornacker. High-Performance Generalized Search Trees, Proc. 24th Int'l Conf. on Very Large Data Bases, Edinburgh, Scotland, September 1999.

[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 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.

[KSF+ 96] Korn F., Sidiropoulos N., Faloutsos C., Siegel E., Protopapas Z.: '*Fast Nearest Neighbor Search in Medical Image Databases*', Proc. 22nd Int. Conf. on Very Large Data Bases, Mumbai, India, pp. 215-226, 1996.

[Kuk 92] Kukich K.: '*Techniques for Automatically Correcting Words in Text*', ACM Computing Surveys, Vol. 24, No. 4, 1992, pp. 377-440.

[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.

[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.

[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.

[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 Similar 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.

[Mum 87] Mumford D.: *'The Problem of Robust Shape Descriptors',* Proc. of the IEEE First International Conf. on Computer Vision, 1987.

[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 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.

[PM 97a] 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.

[PM 97b] Papadopoulos A. and Manolopoulos Y. Nearest neighbor queries in shared-nothing environments. *Geoinformatica* Vol. 1, No. 1, pp. 1-26, 1997(b).

[PM 98] Papadopoulos A. and Manolopoulos Y. Similarity query processing using disk arrays. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1998.

[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.

[RGF 98] Riedel E., Gibson G. A., Faloutsos C. Active storage for large-scale data mining and multimedia. *Proc. 24th Int. Conf. on Very Large Databases*, 1998, 62-73.

[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.

[SAA 00] Stanoi I., Agrawal D., Abbadi A. E.: *'Reverse Nearest Neighbor Queries for Dynamic Databases'*, ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, 2000, pp. 44-53.

[Sag 94] Sagan H.: *'Space Filling Curves'*, Springer-Verlag Berlin/Heidelberg/New York, 1994.

[SBK 92] Shoichet B. K., Bodian D. L., Kuntz I. D.: *'Molecular Docking Using Shape Descriptors',* Journal of Computational Chemistry, Vol. 13, No. 3, 1992, pp. 380-397.

[Sch 91] Schröder M. *Fractals, Chaos, Power Laws: Minutes from an Infinite Paradise*. W.H. Freeman and Company, New York, 1991.

[SH 94] Shawney H., Hafner J.: '*Efficient Color Histogram Indexing*', Proc. Int. Conf. on Image Processing, 1994, pp. 66-70.

[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.

[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.

[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] Stonebraker M., Sellis T., and Hanson E.: *'An Analysis of Rule Indexing Implementations in Data Base Systems'*, Proc. Int. Conf. on Expert Database Systems, 1986.

[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.: *'Satisfying general proximity / similarity queries with metric trees',* Information Processing Letters, p. 145-157, 1991.

[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] Yianilos P.: *'Data structures and algorithms for nearest neighbor search in general metric spaces',* Proc. 4th ACM-SIAM Symposium on Discrete Algorithms (SODA`93), pp. 311-321, 1993.

[Yia 99] Yianilos P.: *'Excluded middle vantage point forests for nearest neighbor search',* Proc. DIMACS Implementation Challenge, ALENEX`99, Baltimore, MD, 1999.

[YY 85] Yao A. C., Yao F. F.: '*A General Approach to D-Dimensional Geometric Queries*', Proc. ACM Symp. on Theory of Computing, 1985.