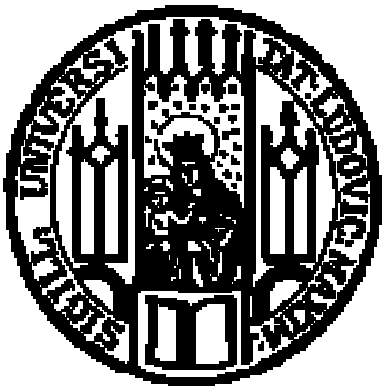
Institut für Informatik

Der

Ludwig-Maximilians Universität

München



Diplomarbeit

The XO-tree

Object oriented design, implementation and evaluation of an index structure for high-dimensional data spaces, based on ovaloid approximation.

Aufgabensteller	:	Prof. Dr. Hans-Peter Kriegel
Betreuer	:	Dr. Christian Böhm
Bearbeiter	:	Stefan Schönauer
Abgabedatum	:	22. Januar 1999

Erklärung

Hiermit erkläre ich, daß ich diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 22.1.1999 .....

Acknowledgements

I would like to thank all the people who supported me during the time I worked on this thesis. First of all Professor Dr. Hans-Peter Kriegel, who made this work possible. The most important person is this context is certainly Dr. Christian Böhm, whose care and support helped through all the ups and downs of this work. Special thanks goes to Gabi Kastenmüller and Gerald Klump. They let the sun shine in the windowless room. Some fruitful discussions about the X-tree took place with Jörn Kohlhammer. Franz Krojer took excellent care of all the technical equipment and always had time for my major and minor problems with it.

Mischa Schirmer and Thomas Trautmann were willing to discuss about this work any time and inspired several important parts of it.

Finally, I would like to thank all my friends and family for their love, care and support, no matter what mood I was in.

Abstract (in German)

Mit der Entwicklung hin zur „Wissensgesellschaft“, gewinnen moderne Datenbanken mit mächtigen Suchwerkzeugen zur Identifikation der relevanten Informationen immer mehr an Bedeutung. Damit einher geht die Erschließung neuer Anwendungsgebiete, wie z. B. CAD, Multimedia, Molekularbiologie und die Analyse von Zeitreihen. Gerade in diesen sogenannten Nicht-Standard-Anwendungen von Datenbanksystemen ist die Ähnlichkeitssuche in großen Datenmengen eine wesentliche Funktion.

Für die Ähnlichkeitssuche werden die Datenobjekte meist mittels einer sogenannten Feature-Transformation in einen hochdimensionalen Vektorraum übertragen. Daher ist der Einsatz einer performanten multidimensionalen Indexstruktur Voraussetzung für eine effiziente Anfragebearbeitung auf diesem Gebiet.

Das Ziel dieser Arbeit war es daher, eine Indexstruktur für solch hochdimensionale Vektorräume zu entwickeln und zu testen. Dazu wurden zunächst bekannte multidimensionale Indexstrukturen untersucht. Dies waren der R^*-Baum, der X-Baum, der SS-Baum, sowie der SR-Baum und der TV-Baum. Es wurde dabei herausgearbeitet, daß die Performanz einer Struktur wesentlich von der Überlappung im Index beeinflußt wird. Die wichtigsten Einflußfaktoren für die Überlappung im Index sind dabei die Form der verwendeten Seitenregion und die Algorithmen zum Aufbau der Struktur.

Im weiteren wurden verschiedene Formen von Seitenregionen mit ihren Vor- und Nachteilen besprochen. Bei der Wahl einer Seitenregion für eine sollte darauf geachtet werden, daß der „tote Raum“ minimiert wird. Damit ist der Teil des Datenraums gemeint, der zwar von der Seitenregion überdeckt wird, jedoch keine

Datenobjekte enthält. Diese Minimierung trägt wesentlich dazu bei, die Überlappung im Index zu verringern. Dabei müssen allerdings zwei Faktoren berücksichtigt werden. Zum einen sollte der Speicherplatzbedarf einer Seitenregion nicht zu groß sein. Andernfalls gerät der entstehende Index zu hoch, was zu einer verminderten Performanz führt. Als zweiter wichtiger Faktor ist die Komplexität einiger Basisoperationen auf Seitenregionen anzuführen. Diese Operationen werden von den Anfragealgorithmen benötigt und sollten daher effizient auszuwerten sein, um Anfragen möglichst schnell beantworten zu können.

Basierend auf diesen Ergebnissen, wurde der XO-Baum entwickelt. Diese neue Indexstruktur fußt auf dem X-Baum, wobei drei neue Formen von Seitenregionen eingeführt wurden. Dies sind Ovaloide, der Schnittkörper aus einem Ovaloid und einem achsenparallelen minimal umgebenden Rechteck und die sogenannte „corner-cut“-Approximation. Sie wurden alle entwickelt, um den toten Raum, der von einer Seitenregion überdeckt wird, zu minimieren und gleichzeitig die Komplexität der Basisoperationen gering zu halten. Zusätzlich wurde der Einfügealgorithmus überarbeitet, um die Minimierung der Überlappung im Index weiter zu optimieren.

Experimente sowohl mit künstlichen, wie mit Realdaten zeigen, daß der XO-Baum für fast alle Anfragetypen ein besseres Leistungsverhalten als der X-Baum zeigt. Im ungünstigsten Fall ist das Leistungsverhalten der beiden Indexstrukturen indentisch. Die Leistungssteigerung liegt meist bei einem Faktor von zwei. Des weiteren zeigen die Experimente, daß der Schnittkörper aus einem Ovaloid und einem achsenparallelen minimal umgebenden Rechteck nicht zur gewünschten Leistungssteigerung führt, wohingegen die anderen beiden neuen Formen von Seitenregionen gleichermaßen guten Ergebnisse für alle Anfragetype liefern. Der XO-Baum erlaubt damit die effiziente Indexierung hochdimensionaler Datenräume, besonders aber solcher mit zehn bis zwanzig Dimensionen.

Table of Contents

1 Introduction

Today's economy depends massively on fast and efficient access to information. It is undoubted that this dependency will even increase in the future. The problem, however, is not so much the collection of information, as a vast amount of data is produced everyday in each enterprise. What is needed, are tools to find the relevant information for a specific task effectively and efficiently.

If the information that is searched is of simple structure, like one-dimensional numerical attributes or character strings, there exist widely accepted solutions to the problem. The index structures provided by database management systems (*DBMS*), like the B^+-tree [BM 77], are well suited for this type of data.

In recent years, the number of applications that need to process large numbers of complex data objects is growing rapidly [Jag 91, FRM 94, Ber 97, Kei 97, Sei 97]. In application domains such as multimedia, medical imaging, molecular design, computer aided design etc., complex data objects are common. In these, so called *non-standard databases*, the search is often driven by some notion of similarity rather than the exact match of objects. Nevertheless, efficient search tools are essential there, due to the enormous and even increasing size of such databases.

This makes index structures necessary, which are specifically designed for high-dimensional data spaces to index the complex and high-dimensional data objects in non-standard databases. Such an index structure will be developed in the forthcoming chapters. First, however, we will examine a few applications using non-standard databases and inspect some of the requirements they impose on the DBMS. Afterwards some prerequisites for the similarity search process will be discussed.

1.1 Applications using Non-Standard Database Systems

1.1.1 Similar Geometric shapes in CAD Databases

Most current Computer Aided Design (CAD) systems are file based and do not use any database technology. Some modern CAD systems, however, use object-relational or object-oriented database technology to store the objects. While this supports data independence, concurrency and recovery, the search facilities are limited to simple operations like the retrieval of an object according to its key.

The S3-System (Similarity Search System), developed in a recent research project [Ber 97, BK 97, BKK 97], provides a powerful tool for the search of similar parts. The scope of the project was to reduce the diversity of parts, namely plastic clips, in the car industry and avoid the redesign of parts for which similar designs already exist. The avoidance of redesigns bears a great cost saving potential because mounting tools and injection moulds can be reused and manpower can be saved. The objects in the S3-projects were two-dimensional. Consequently, another application domain for the search of similar shapes uses similar techniques: computer vision [Jag 91, GM 93, MG 93].

A distinctive measure between the applications is the way they define the similarity of two objects. These definitions differ not only in the way they are derived, but also in their properties. Consequently, different similarity measures yield often different invariances, which may be meaningful and desired in one context, but meaningless or even unwanted in others. Here a few of the invariances that are usually considered as important:

- Translation invariance
- Rotation invariance
- Invariance with respect to uniform and non-uniform scaling
- Shearing invariance
- Invariance with respect to partial object occlusion

Moreover, a distinction between *partial and total similarity* can be made. In this context, *total similarity* means that two objects are over all similar, where in *partial similarity* the objects only have to be similar in some detail.

In the S3-project, similarity for two-dimensional polygons is defined in two different ways. In the first definition, *section coding* is used, which is based on volume coincidence. Starting at the center of gravity, the object is cut into pieces like a cake (cf. figure 1). The relative volume of each piece that is covered by the object is determined and the vector of all these ratios is used as a feature vector to determine the similarity of two objects. Seidl and Kriegel [Sei 97, KKS 98] extend this model by using the more general quadratic form distance instead of the Euclidean metric for the determination of the feature distance (cf. section 1.1.2). This way, vicinity properties of vectors can be taken into account, making the model more realistic. Section coding is invariant with respect to scaling, translation and (to a limited degree) rotation.

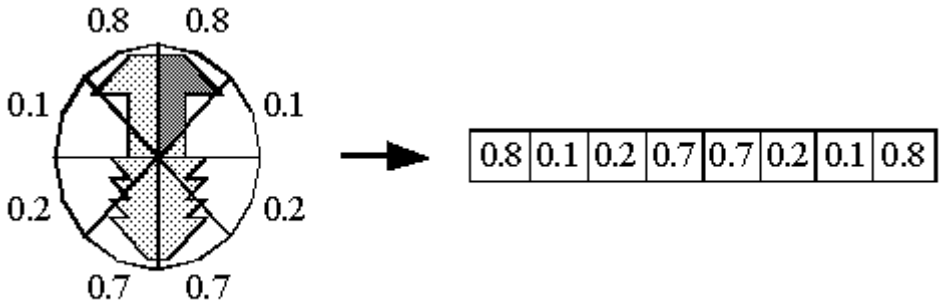


Figure 1: Section Coding.

The boundary of the polygon forms the basis for the second similarity measure of the S3-project. The line segments of the polygon are transformed into a parametric form by determining their curvature. The coefficients of the Fourier transformed parametric form are then, again, interpreted as vectors in Euclidean space. Total similarity is defined by applying this technique to the entire polygon. To determine partial similarity, the object is separated into sequences of line segments with fixed length. The parameterization and the Fourier transformation are then applied to each of the sequences separately.

Jagadish [Jag 91] uses a different technique to define the similarity of two-dimensional shapes. The basis for this is a rectilinear cover of the object, i.e. a cover consisting of axis-parallel rectangles (cf. figure 2). The rectangles covering the object are sorted by size and the largest ones are used as key for an index. Due to normalization prior to the determination of the cover, this technique achieves invariance with respect to scaling and translation, but is not rotation invariant.

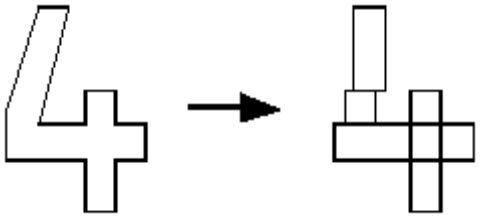


Figure 2: Rectangular Cover of an Object [Jag 91].

Another method is used by the QBIC (Query By Image Content) system [FBFH 94], which also contains a component for 2-D shape retrieval. The method, which expects shapes as sets of points, is based on algebraic moment invariants and can be used for 3-D objects [TC 91], too. Invariance with respect to rigid transformations (translations and rotations) is inherent to this method. This is an important feature in a CAD database. Nevertheless, there are few possibilities to adjust the method to specific application domains. From all the available moment invariants, the appropriate ones have to be chosen and their weighting factors may be changed.

1.1.2 Similarity of Color Images Based on Histograms

The QBIC system just mentioned defines the similarity of two color images based on the color distribution in the pictures. Two images are defined as similar, if they contain the same colors with similar frequency. The distribution of the colors is called *color histogram*. The histogram of a picture is obtained by determining for each color, the ratio of pixels with that specific color. This takes place, after the color spectrum has been reduced and normalized to a manageable size (cf. figure 3).

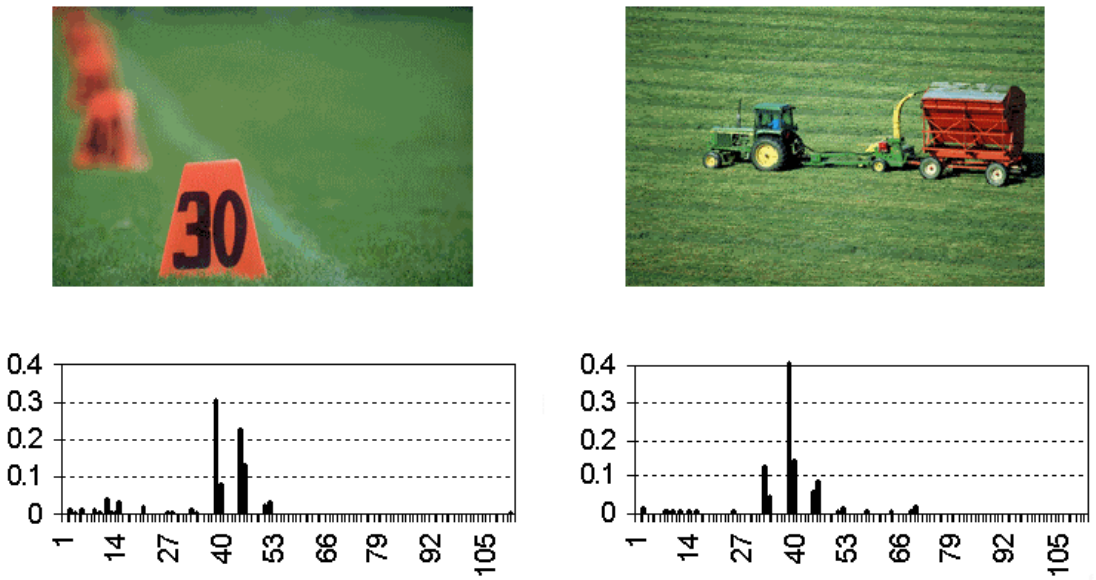


Figure 3: Two similar images and corresponding 112-D color histograms [Sei 97].

Seidl and Kriegel [SK 97] also use histograms as feature vectors. They do not define the similarity of two histograms as their Euclidean distance. Although this approach seems natural, it would result in the difficulty that all pairs of different colors are seen as equally dissimilar. To a human, however, some colors are rather similar (e.g. red and orange) whereas others are very dissimilar (e.g. red and green). This can be taken into account, if not the Euclidean distance between two histogram vectors is determined, but the following quadratic form distance metric is used instead:

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

Here, the similarity matrix A contains the information, which colors are similar to each other and in what degree. With this similarity matrix, the method can easily be adjusted to the specific needs of an application domain. Both, the QBIC system [FBFH 94] and Seidl and Kriegel [SK 97] use the quadratic form distance metric to define the similarity of color images.

1.1.3 Molecular Biology

Molecular biology is another field where similarity queries are of great importance. Most biological functions in organisms are performed by the interaction of proteins. For the function of a protein, its three-dimensional structure is the defining property, i.e. proteins with a similar geometrical structure usually have a similar function.

With this in mind, the prediction of molecular interaction obviously becomes an interesting and important challenge. Two molecules interact, if they have a complementary surface structure with respect to geometric shape and electromagnetic and chemical properties. Apparently, finding a molecule with a complementary structure is closely related with the similarity search problem. The method of choice is simply to build the complement of the query object and then retrieve all objects from the database, which are similar to the complementary object.

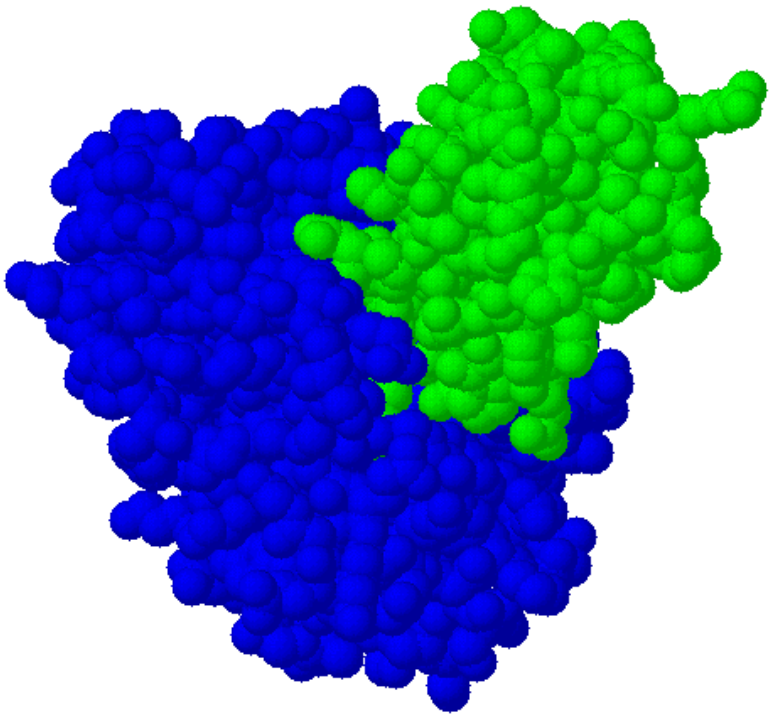


Figure 4: Two Docking Proteins [Sei 97].

Kriegel, Schmidt and Seidl [KSS 97, KS 98] define the similarity of two molecule surfaces by fitting standard segments, e.g. paraboloids, to the molecular surface and then determine the approximation error. The mutual approximation error is then used to measure the (dis-)similarity of the two molecules. They use the three-dimensional structures of molecules as provided by the Brookhaven Protein Data Bank, which contains more than 3000 molecules.

1.1.4 Time Sequence Analysis

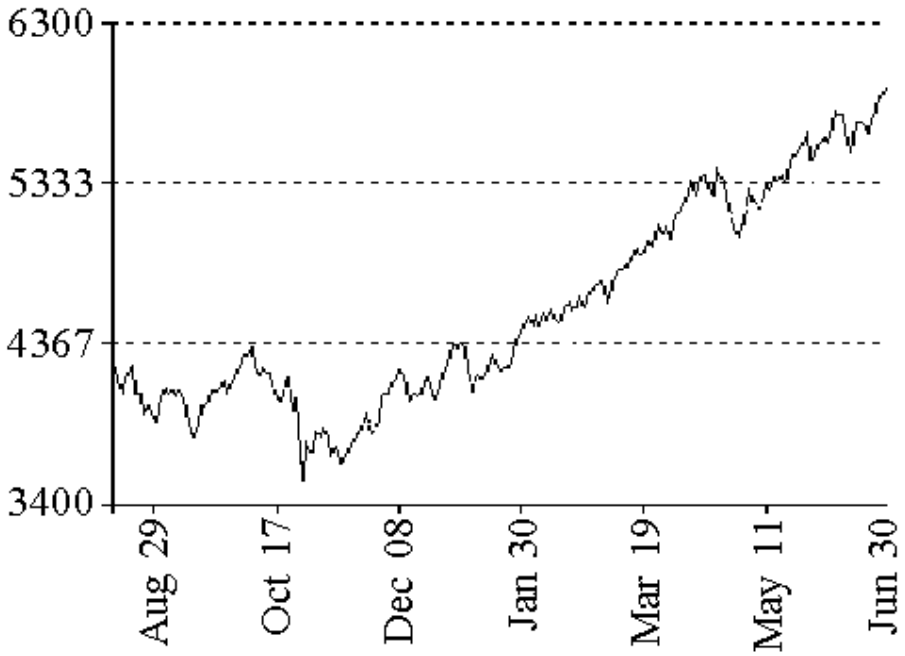


Figure 5: DAX Performance Index (Source: Frankfurt Stock Exchange).

Time sequence analysis is especially important in economic, but also in other sciences. It is often used in data mining applications to:

- Determine products with similar selling patterns
- Identify customers with similar buying behavior
- Discover stocks with similar price movements (cf. figure 5)
- Identify populations with a similar development in size

Time sequence analysis can be divided into two categories. The first is *whole matching*, where the entire sequences, which must have the same length *n*, have to be similar. With *subsequence matching*, the query sequence is smaller and a subsequence of a larger sequence, which matches the query sequence best, is searched.

Agrawal, Faloutsos and Swami present a method for whole matching of one-dimensional sequence data in [AFS 93]. They use the square root of the sum of squared differences between two sequences *x* and *y* as similarity measure:

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

This corresponds with the Euclidean distance of vectors and with the energy of the difference signal in a signal theoretic sense. To obtain vectors of a reasonable size, the sequences are mapped to a low-dimensional feature space using the Discrete Fourier Transform.

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

1.2 Feature Transformation

The notions of similarity in the applications above seem rather different from each other. However, they have a few properties in common, which allow the use of the same indexing and query techniques for all of them.

1.2.1 Object Distance

An important community of the similarity measures in all the sample applications is the fact that similarity is defined in terms of a distance between two objects. Consequently, a similarity measure δ assigns a positive value to a pair of objects, which is a measure for the similarity of the two objects:

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

The higher δ is for a pair of objects, the less similar are those objects. Generally δ is chosen so that it yields zero if and only if the two objects are identical, although this may not be necessary in some applications. Due to its properties, δ is called the *object distance*. In all the applications above, δ fulfills the requirements for a metric, as it is positive, symmetric and fulfills the triangle inequality. This lead to the development of several query processing techniques, which can handle objects in a metric space directly, in recent time [Yia 93, Chi 94, CPZ 97]. Nevertheless, none of these structures was applied in any of the sample applications, because these structures lack the performance required by those applications.

1.2.2 Feature Distance

Usually, operations on the real objects are very costly, because of the complexity of the object description. To avoid this and achieve an efficient similarity query processing, a so-called *feature transformation* is applied. This means that important features of the objects in the database are extracted and transformed into d-dimensional vectors in a vector space (*feature vectors*). The feature transformation is defined in a way that the distance between the feature vectors, the *feature distance*, either corresponds with the object distance or is at least a lower boundary to this distance. Consequently, the similarity search can easily be expressed as a range query in the feature space.

The feature transformation has to extract the most important and distinguishing properties of the objects in order to yield a good query performance. Therefore, it will usually be provided by an expert in the corresponding application domain. In the S3-project two different feature transformations were used: section coding and Fourier transform of line segments. In the molecular biology example, the features were extracted by approximating the objects using standard surfaces as paraboloids. Color histograms were the feature vectors of the image database systems and the Discrete Fourier transform was used in the time sequence database example. One can proof that in all these cases, the feature distance is a lower bound of the objects distance. This is a necessary condition to avoid false dismissals and ensure the correctness of the query algorithms.

1.2.3 Multi-Step Query Processing

If the feature distance does not directly correspond with the object distance, but is only a lower bound, ambiguities may be introduced. To handle these ambiguities a *multi-step query processing* is necessary. This query processing consists of two parts. In the so-called *filter step*, a range query on the feature space is processed. The result of this is a set of candidates. As the feature distance is a lower bound of the object distance, it is guaranteed that each object contained in the query range is also in the candidate set (*no false dismissals*), but not every object in the candidate set has to be an actual answer to the similarity query. Therefore, the objects in the candidate set have to be tested in object space if they are part of the answer set. This is done in the next phase, which is named *refinement step*. Obviously, this multi-step query

processing yields a good performance only if the *filter selectivity* is good, i.e. only a few candidates have to be tested in object space.

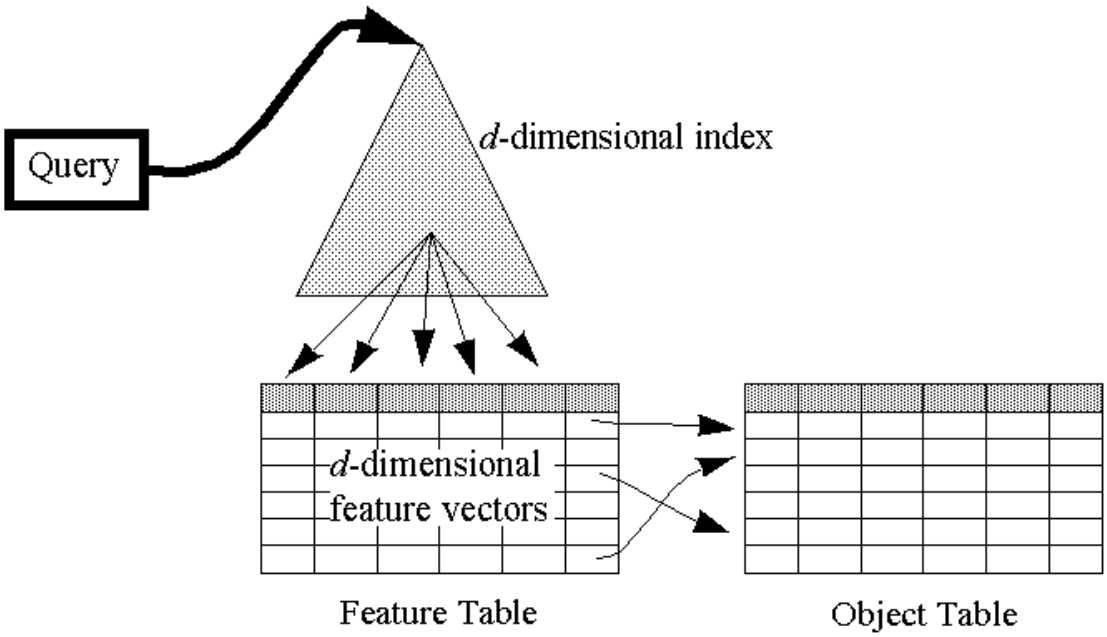


Figure 6: Multi-Step Query Processing of Similarity Queries.

Figure 6 shows the entire multi-step query processing. An index of the feature vectors is used to process the range query of the filter step. Afterwards, the objects in the candidate set have to be loaded and the false hits are excluded in the refinement step. The algorithm for the refinement step is application specific and will not be considered here. The filter step, however, is identical for any application. It should therefore be supported by a database management system. Consequently, we will concentrate on query processing in feature space from now on.

1.2.4 Index Structures

For the performance of the filter step, appropriate support by a multi-dimensional index structure is crucial. For data spaces with sufficiently small dimension, e.g. 3, index structures like the grid file [NHS 84], the kd-tree [Ben 75, Ben 79] or the R^*-tree [BKSS 90] can be used. However, these index structures show an insufficient performance if the dimension is high, e.g. 16. The term *'curse of dimensionality'* is used to describe this problem. It is based on the fact that most measures one could define in a vector space, e.g. volume or perimeter, depend

exponentially on the dimension of the vector space. Therefore, many approaches work well only in low-dimensional data spaces where the exponent is small enough. Unfortunately, in many applications indexes for high-dimensional data spaces are needed. This led to the development of several index structures specifically designed to deal with the problems in high-dimensional spaces, like the SS-tree [WJ 96] or the X-tree [BKK 96].

2 Related Work

Numerous different tree structures were proposed for the indexation of multi-dimensional data. They can be divided into two classes. On the one hand there are data organizing structures such as R-trees [Gut 84, BKSS 90] on the other hand the space organizing structures such as Multidimensional Hashing [HSW 88a, KS 86, KS 87, Oto 84] or grid-files [NHS 84, Fre 87, Hin 85, HSW 88b, KW 85, KS 88, Ouk 85]. As hashing-based structures do not play an important role in multidimensional indexing, only members of the first class will be discussed. In this chapter, a collection of them, which influenced the development of the XO-tree, will be presented. We start with some basic definitions and a short introduction on tree structures and query types. The R^*-tree that follows forms the basis for all the other index structures presented. The X-tree was the basis for the XO-tree structure developed in the next two chapters. After the SS-tree with its different approximation technique, the SR-tree represents a similar approach as the XO-tree. The chapter closes with a brief description of the TV-tree and its main concepts.

2.1 Trees and Queries

2.1.1 Basic Definitions

We need to introduce some basic notions in order to formalize the forthcoming descriptions. First, we need to define our notion of a database.

In this context we assume that all objects are feature-transformed into points of a vector space with a fixed, finite dimension d. This makes a database a set of points in a d-dimensional data space DS, which in turn is a subset of $\Re^d$. Analysis as well as

the implementation are greatly simplified if the data space is restricted to the unit hyper cube: $DS = [0..1]^d$.

Our database is completely dynamic in a sense that insertions and deletions of points are possible and should be handled efficiently. The number of points currently stored in the database is abbreviated to n. Here we must mention that the term *point* is ambiguous. Sometimes it stands for a point object, i.e. a point stored in the database, in other situations we mean a point in the data space, i.e. a position which is not necessarily stored in the database. An example for the latter case is the query point. From the context, the meaning of the notion point will always be obvious.

Definition 1: Database

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

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

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

$$DS \subseteq \Re^d.$$

In some applications, it is not possible to map objects into feature vectors, but there exists some notion of similarity between objects that can be expressed as a metric distance between the objects. These distances are then used for query evaluation. Several index structures for such metric spaces have been proposed [CPZ 97, Yia 93, Chi 94, Uhl 91, Bri 95, BO 97]. Fully aware of this problem we restricted our definition of database to vector spaces with finite dimension and therefore we will not consider these approaches.

Neighborhood queries are based on the notion of the distance between two points *P* and *Q* in the data space. Which distance metric is used depends on the application that has to be supported. Certainly, the most common metric is the Euclidean metric L_2 defining the well-known Euclidean distance function δ_{em}:

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

Other widely used L_p metrics are the Manhattan or *city block metric* L_1 and the maximum metric L_∞:

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

Whereas queries using the L_2 metric are (hyper-)sphere shaped, those using the maximum or the Manhattan metric are hypercubes and rhomboids, respectively (cf. Figure 7).

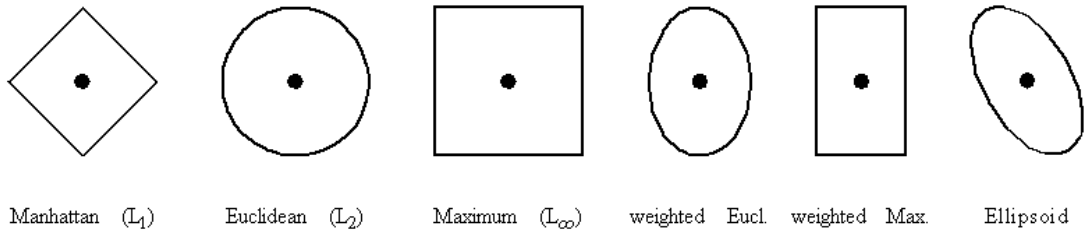


Figure 7: Query Shapes for Different Metrics.

Assigning *weights* $w_0, ..., w_{d-1}$ to the dimensions to defines the weighted Euclidean and the weighted Maximum metrics, which correspond to axis-parallel ellipsoids and axis-parallel hyperrectangles:

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

2.1.2 Trees

The data organizing index structures are based upon the principle of hierarchical clustering of the data space, which makes them structural similar to the B^+-tree [BM 77, Com 79]. Data vectors are stored in data nodes in a way that spatially adjacent vectors are likely to be in the same node. As each data vector is stored in exactly one data node, there is no object duplication among the data nodes, except for different objects with identical data vectors. The data nodes themselves are organized in a hierarchical directory, where each directory node points to a set of subtrees. Therefore, these subtrees either are data nodes or headed by a directory node themselves. The directory nodes usually have different internal structure than the data nodes. On top of the structure is one single directory node, which is called

the *root node*. The root node is usually the only node that is allowed to be less than minimally filled. Each query and update processing starts at this node. The tree structures are all height balanced as all paths from the root to all data pages have the same length. This length is called the *height* of the index and may only change after an insert or delete operation. The length of the path from the root to a node is the *level* of this node, with the root itself having level one.

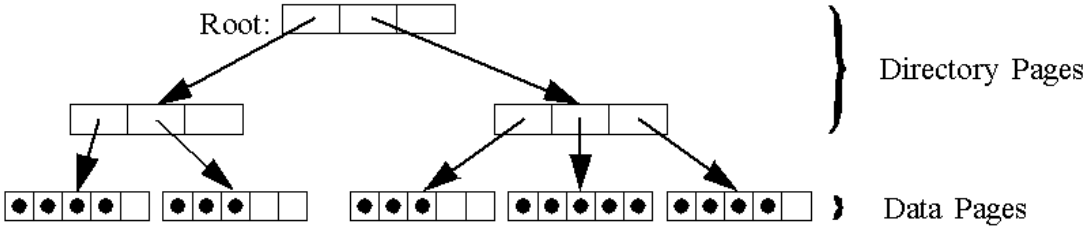


Figure 8: Tree Structure.

High-dimensional access methods are primarily designed for secondary storage use. Therefore, data nodes are stored in data pages with a capacity of $C_{max,data}$, defining the maximal number of data vectors that can be stored in one data page. Analogously, the directory page capacity $C_{max,dir}$ represents an upper limit for the number of subnodes referenced in each directory page. Originally $C_{max,data}$ and $C_{max,dir}$ were chosen so that data and directory pages fitted exactly into one page of the secondary storage. Nowadays, however, the size of a page on a disk is seen as a hardware detail, which should be hidden from the programmer and user. But reading larger consecutive blocks from a disk is still orders of magnitude faster than reading smaller blocks at random positions. Therefore $C_{max,data}$ and $C_{max,dir}$ represent user-defined logical page sizes for an artificial paging. Although interesting, the optimal choice of these values will not be content of this work. Normally the logical page size is constant. The X-tree and the XO-tree, however, allow nodes to span over multiples of the basic page size. This is called the supernode concept and will be discussed more thoroughly below.

As all the index structures presented are dynamic, i.e. they permit insert and delete operations in *O(log n)* time, they allow nodes to be filled below their maximum capacity C_{max}. Nevertheless, they require all nodes except the root node, to be at least

filled up to a minimal value. This threshold is called the *minimum storage utilization* su_{min}, and usually is about 40% of C_{max}.

As mentioned above, high-dimensional index structures cluster the data in order to store spatially close objects into the same node, if possible. This is achieved by assigning each page a certain region, which is a subset of the data space. This region is called *page region* and can be of arbitrary shape, but is always completely enclosed by the region of its parent node. Consequently, all objects stored in a subtree are always contained in the page region of the root of this subtree. This makes a page region a *conservative approximation* for the data objects and other page regions stored in the corresponding subtree (cf. chapter 3). For most index structures, the page regions may overlap, although this leads to performance losses and therefore should be avoided or at least minimized, whenever possible. In query processing, the page region is used to exclude branches of the tree from further processing. For efficient query processing it is therefore essential that the test for intersection with a query region and the computation of the distance to the query region, in case of a nearest neighbor query, can be performed efficiently. For a more thorough discussion of page regions refer to chapter 3.

2.1.3 Queries

Queries are by far the most important operations on index structures. Therefore optimizing the query performance of a structure is an important design step. In order to fulfill this task, we first have to investigate which types of queries are of relevance in high-dimensional data spaces.

The exact match query is essential for all kinds of data spaces. It is defined as follows: Given a query point *q*, determine if *q* is contained in the database or not. Query processing starts at the root node that is loaded into the main memory. For all page regions containing *q*, the function ExactMatchQuery is called recursively with *q* and the address of the corresponding page as parameters. Since all index structures presented here allow overlapping page regions, it is possible that several sons of the current node have to be visited to determine the result of the exact match query. The result is true if a data page is found with a point stored in it that matches the query point.

The point query is a generalization of the exact match query. It retrieves all points from the database which have the identical coordinates as the query point. Therefore an exact match query returns true if the cardinality of the result set of the corresponding point query is greater than zero.

Definition 2: Point Query

For a given query point *Q* in the data space the point query retrieves the following set from a database *DB*:

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

The pseudocode for a point query is shown in figure 9.

```
PointSet PointQuery(Point q, PageAdr pa)
{
    int i;
    PointSet result = EmptyPointSet;
    Page p = LoadPage (pa);
    if (IsDataPage (p))
        for(i = 0; i < p.num_objects; i++)
            if (q == p.object[i])
                AddToPointSet(result, p.object[i]);
    if (IsDirectroyPage (p))
        for(i = 0; i < p.num_objects; i++)
            if (IsPointInRegion(q, p.region[i]))
                PointSetUnion(result, PointQuery(q, p.childpage[i]);
    return result;
}
```

Figure 9: Algorithm for Point Queries.

Another important query type is the range query, which returns a set of points contained in the query range. This implies that the result set is of unknown size and may even consist of the entire database. The algorithm presented here is formulated independently from the applied metric (cf. section 2.1.1). As long as effective and efficient tests for the two predicates IsPointInRange and RangeIntersectRegion are provided, every metric is applicable, including such with weighted dimensions. With this in mind, also partial range queries, i.e. queries where only a subset of the attributes is specified, can be processed like normal range queries. Partial range queries are then just regular range queries with the unspecified attributes weighted

zero. In addition, window queries can be expressed as range queries by using a weighted L_{max} metric.

Definition 3: Range Query

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

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

This definition shows that even point queries can be expressed as range queries by setting the query range *r* zero.

```
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 (IsDirectroyPage (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 10: Algorithm for Range Queries.

A last very important query type in high-dimensional data spaces is the nearest neighbor query and its generalization, the k-nearest neighbor query. This query type overcomes a disadvantage of the range query and its special cases (point and window query): the size of the result set is unknown. This makes it sometimes very hard for the user to specify the query range. Consequently, the answer he gets is one of two extremes: either he gets no answers at all or almost the entire database is returned as result. Especially when the similarity of objects is the query criterion (‘Find the most similar object to the query object.’), normally only the result set size is known but not the range in which the results lie. Therefore, the nearest neighbor query and the k-nearest neighbor query are introduced. A nearest neighbor query returns the closest

object to the query object in the database with respect to a given distance metric. Of course, it is possible that several objects have the same distance to the query object. To resolve this tie situation we use non-determinism and choose an arbitrary point from the set of all such closest points in our definition.

Definition 4: Nearest Neighbor Query

For a given query object *Q* and a given distance metric *M*, a nearest neighbor query retrieves the following set from a database *DB*:

$$NNQuery(DB, Q, M) = SOME\left\{P \in DB \middle| \forall P' \in DB : \delta_M(P,Q) \leq \delta_M(P',Q)\right\}$$

If not only the closest point is wanted but rather a natural number *k* of closest points, the more general *k-nearest neighbor query* is used. This query retrieves the k closest points to the query point from the database. Of course, the same problem with tie situations arises and again the use non-determinism can be used to solve the problem.

Definition 5: k-Nearest Neighbor Query

For a given query object *Q*, a natural number *k* and a distance metric *M*, a k-nearest neighbor query retrieves the following set from the database *DB*:

$$kNNQuery(DB, Q, k, M) = \left\{ P_0 \ldots P_{k-1} \in DB \middle| \begin{array}{l} \neg\exists P' \in DB \setminus \{P_0 \ldots P_{k-1}\} \wedge \\ \neg\exists i, 0 \leq i \leq k : \delta_M(P_i,Q) > \delta_M(P',Q) \end{array} \right\}$$

From the two approaches to process nearest neighbor queries only the one published by Hjaltason and Samet [HS 95] (‘*HS algorithm*’) will be described. The algorithm presented by Roussopoulos, Kelley and Vincent [RKV 95] (‘*RKV algorithm*’) is described in greater detail in [Böh 98]. A thorough comparison between the two is presented there, too. It reveals that the HS algorithm is optimal in terms of page accesses and is a lot easier to extend for k-nearest neighbor queries. Even ranking queries, where the user is able to ask for the next closest objects after getting the nearest neighbor, can easily be processed. It should not be omitted that the HS algorithm has a worst-case space complexity of *O(n)* and therefore might not be suitable for some applications. Nevertheless, it was chosen for the processing of nearest neighbor queries in the sample implementation of the XO-tree.

The HS algorithm accesses the pages of a tree in order of increasing distance to the query point, starting at the root node. This requires that the algorithm is allowed to jump between branches and levels of the tree. For this, an active page list (APL) is maintained which stores the background storage address and the distance to the query point of all active pages. A page is called *active* if its parent node has been processed, but not the page itself. Since the parent of an active page has already been loaded, the corresponding region of all active pages is known and their distance to the query point can be determined. As only the distance between an active page and the query point is needed, the representation of the page regions is not stored in the APL. The APL is implemented as a priority queue with the distance to the query point as sorting parameter. In the worst case, it contains entries for all pages of the index. Therefore, an implementation of a priority queue, which is suitable for secondary storage will normally be needed.

```
/* HS algorithm */
Point NearestNeighborQuery(Point q, Metric m, PageAdr root)
{
    int i;
    Element e = LoadPage(root);
    PriorityQueue APL = new PriorityQueue();
    Push(APL, e, dist(e, q, m));
    while (IsNotEmpty(APL))
    {
        e = Pop(APL);
        if (IsPoint(e))
            return e;
        if (IsDataPage(e))
            for (i = 0; i < e.num_objects; i++)
                Push(APL, e.object[i], dist(e.object[i], q, m));
        else /* Element is a directory page */
            for (i = 0; i < e.num_objects; i++)
                Push(APL, e.childpage[i], dist(e.childpage[i], q, m));
    }
}
```

Figure 11: HS-Algorithm for Nearest Neighbor Queries.

The algorithm in pseudocode is shown in figure 11. Obviously for k-nearest neighbor queries there is a second priority queue with fixed length k needed to hold the closest point candidate list. By storing the APL, it is easily possible to retrieve the next closest objects in ranking queries.

2.2 The R*-tree

The R*-tree [BKSS 90], a multidimensional index structure for rectangle and point data, is the most successful variant of the R-tree [Gut 84].

Like the R-tree, the R*-tree uses axis parallel minimal bounding rectangles (MBR's) as page regions. But Beckmann, Kriegel, Schneider and Seeger carefully studied the R-tree algorithms under various data distributions and then identified further optimization objectives in addition to the optimization for small volume of the page regions, which Guttman uses. They minimize the overlap between page regions as well as their surface and the volume covered by internal nodes. At the same time, maximal storage utilization is wanted. This is achieved by two major changes to the R-tree insertion processes. First, the heuristic for choosing a suitable page for the insertion of a new object is modified. Additionally the concept of forced reinserts is introduced.

```
PageAdr Insert(Point object, PageAdr pa)
{
    Page p = LoadPage(pa);
    PageAdr subtree;
    PageAdr new_son;
    PageAdr brother = NULL;
    if (IsDataPage(p))
        InsertObjectInPage(object, p);
        if (Overflow(p))
            brother = OverflowTreatment(p);
    if (IsDirectoryPage(p))
        subtree = ChooseSubtree(p, object);
        new_son = Insert(object, subtree);
            if (new_son)
                InsertSonInPage(new_son, p);
                if (Overflow(p))
                brother = OverflowTreatment(p);
    return brother;
}
```

Figure 12: Insert-Algorithm of the R*-tree.

When an object is inserted into an R*-tree, one of three cases may occur on every level of the tree during the search for a suitable page into which the object can be inserted. If exactly one page region contains the object, the corresponding node of this region is used.

```
PageAdr ChooseSubtree(Page p, Point object)
{
    int i;
    PageAdr subtree;
    float MINOverlEnl = INFINITY, MINVolEnl = INFINITY;
    float MINVol = INFINITY;
    float OverlEnl, VolEnl, Vol;
    if (IsDataPage(p.childpage[0]))
    {
        /* Determine minimum overlap enlargement! */
        for (i = 0; i < p.num_objects; i++)
        {
            OverlEnl = OverlapEnlarge(p.childnode[i], object);
            VolEnl = VolumeEnlarge(p.childnode[i], object);
            Vol = Volume(p.childnode[i]);
            if ((OverlEnl < MINOverlEnl) || (OverlEnl == MINOverlEnl &&
            VolEnl < MINVolEnl) || (OverlEnl == MINOverlEnl &&
            VolEnl == MINVolEnl && Vol < MinVol))
            {
                MINOverlEnl = OverlapEnlarge(p.childnode[i], object);
                MINVolEnl = VolumeEnlarge(p.childnode[i], object);
                MINVol = Volume(p.childnode[i]);
                subtree = p.childnode[i];
            }
        }
    }
    else
    {
        /* Determine minimum volume enlargement! */
        for (i = 0; i < p.num_objects; i++)
        {
            VolEnl = VolumeEnlarge(p.childnode[i], object);
            Vol = Volume(p.childnode[i]);
            if ((VolEnl < MINVolEnl) || (VolEnl == MINVolEnl &&
            Vol < MINVol))
            {
                MINVolEnl = VolumeEnlarge(p.childnode[i], object);
                MINVol = Volume(p.childnode[i]);
                subtree = p.childnode[i];
            }
        }
        return subtree;
    }
}
```

Figure 13: ChooseSubtree-Algorithm of the R*-tree.

When several different pages contain the object, the one with the smallest volume is chosen. The change compared to the R-tree is in the third case, when the object is not

contained in any page region. In this case, a page region has to be chosen, which must be adapted afterwards. Here the R*-tree makes a distinction whether the child node of the current node is a data or a directory page.

If it is a directory page, the region with the smallest volume enlargement is chosen. In case of an ambiguity, the region with the smallest volume is taken.

If the child node is a data page, the region, which yields the smallest enlargement in overlap, is chosen. Further criteria in tie situations are the enlargement in volume and the volume itself. The algorithm shown in figure 13 implements these optimization goals, although it does not explicitly distinguish between the three cases.

If a page overflow occurs during the insertion process, prior to a split a forced reinsert takes place. This means that a defined percentage (usually 30%) of the objects with the highest distance to the center of the region is deleted from the node and the region is adapted. The deleted entries are then reinserted into the index. One of the advantages of this behavior is that splits can often be avoided. Additionally, the quality of the partitioning improves, as unfavorable decisions in the beginning of the index construction can be corrected this way. Besides this, the average storage utilization grows to a factor between 71% and 76%.

```
PageAdr OverflowTreatment(PageAdr pa)
{
    PageAdr brother = NULL;
    if (!IsRoot(pa) && IsFirstOverflow(Level(pa)))
            ReInsert();
    else
        brother = Split(pa);
    return brother;
}
```

Figure 14: Overflow Treatment of the R*-tree.

If necessary a two-phase split is performed. In the first phase, the split axis is chosen. For this, the objects are sorted according to their lower bound and according to their upper bound in every dimension. For each of these sortings, a number of partitionings with controlled degree of asymmetry is observed. The surface area of the MBR's of all partitionings in one dimension is summed up and the dimension

with the least sum is chosen as split axis. The pseudocode for this algorithm is depicted in figure 15.

```
int ChooseSplitAxis(Page p)
{
    int i, splitaxis;
    Sorting Low, High;
    Distribution LowDistrib, HighDistrib;
    float MINMarginSum = INFINITY;
    float MarginSum;
    /* determine splitaxis */
    for (i = 0; i < dimension; i++)
    {
        Low = SortByLowerValue(p.childnode, i);
        High = SortByHigherValue(p.childnode, i);
        LowDistrib = ComputeDistributions(Low);
        HighDistrib = ComputeDistributions(High);
        MarginSum = MarginSum(LowDistrib, HighDistrib);
        if (MarginSum < MINMarginSum)
        {
            MINMarginSum = MarginSum;
            splitaxis = i;
        }
    }
    return splitaxis;
}
```

Figure 15: Choice of the Split-Axis in the R*-tree.

In the second phase, the split plane is determined. The main goal in this phase is the minimization of the overlap between the page regions. In case of any ambiguities the least coverage of dead space, i.e. data space with no objects in it, is used as criterion. Again, only a limited number of partitionings, which are created analogously to those in the first phase, is observed. For the pseudocode refer to figure 16.

The R*-tree was tested on various distributions of rectangle data. It showed performance improvements between 10% and 75% over the R-tree. However, Berchtold, Keim and Kriegel studied the R*-tree split algorithm in higher-dimensional data spaces in [BKK 96]. They found that the algorithm leads to deteriorated directories with a high overlap in high-dimensional data spaces, which makes it necessary to load the entire index in order to process most queries. Therefore, the R*-tree is not adequate for these data spaces.

```
PageAdr Split(PageAdr pa)
{
    int i, splitaxis;
    Page p = LoadPage(pa);
    Sorting Low, High;
    Distribution LowDistrib, HighDistrib;
    float MINOverlap, MINVolume;
    PageAdr new_brother;

    /* determine splitaxis */
    splitaxis = ChooseSplitAxis(p)

    /* determine split index */
    Low = SortByLowerValue(p.childnode, splitaxis);
    High = SortByHigherValue(p.childnode, splitaxis);
    LowDistrib = ComputeDistributions(Low);
    HighDistrib = ComputeDistributions(High);
    for (i = 0; i < NumberOfDistributions; i++)
    {
        if ((Overlap(HighDistrib[i]) < MINOverlap) ||
        (Overlap(HighDistrib[i]) == MINOverlap &&
        Volume(HighDistrib[i]) < MINVolume;
        {
            MINOverlap = Overlap(HighDistrib[i]);
            MINVolume = Volume(HighDistrib[i]);
            SplitDistrib = HighDistrib[i];
        }
        if ((Overlap(LowDistrib[i]) < MINOverlap) ||
        (Overlap(LowDistrib[i]) == MINOverlap &&
        Volume(LowDistrib[i]) < MINVolume;
        {
            MINOverlap = Overlap(LowDistrib[i]);
            MINVolume = Volume(LowDistrib[i]);
            SplitDistrib = LowDistrib[i];
        }
    }
    new_brother = SplitAccordingTo(SplitDistrib);
    return new_bother;
}
```

Figure 16: Split-Algorithm of the R*-tree.

2.3 The X-tree

As just mentioned, empirical studies [BKK 96, WJ 96] show deteriorated performance for high-dimensional data in the R*-tree. This can not be explained simply by a lower fanout but vastly depends on the fact that the overlap in the directory increases rapidly with the dimension. Unlike in low-dimensional spaces, the freedom of choice for a split axis is extremely limited. Often there is only one split axis with the desired properties. If an index structure does not choose this axis, it will produce high overlap between the MBR's in a directory page, thus showing massive performance losses in high-dimensional data spaces. Unfortunately, this split axis might lead to an unbalanced partition of the data space. In these cases, it is favorable not to split the node at all, to guarantee minimum storage utilization.

The X-tree [BKK 96] was specifically designed to avoid these problems. It extends the R*-tree by the two concepts of an *overlap-free split* and the introduction of so-called *supernodes* with an enlarged page capacity. The overlap-free (only overlap-minimal for extended objects) split is based on a split-history. This split-history is created if one records the history of all page splits in an R*-tree. During the creation of an index, one starts for example with a data page A covering almost the entire data space and inserts data into it. Eventually the page overflows and is split into the two new pages A' and B perpendicular dimension 2. Each of those pages might be split again later on. This results in a binary tree similar to the one depicted in figure 17.

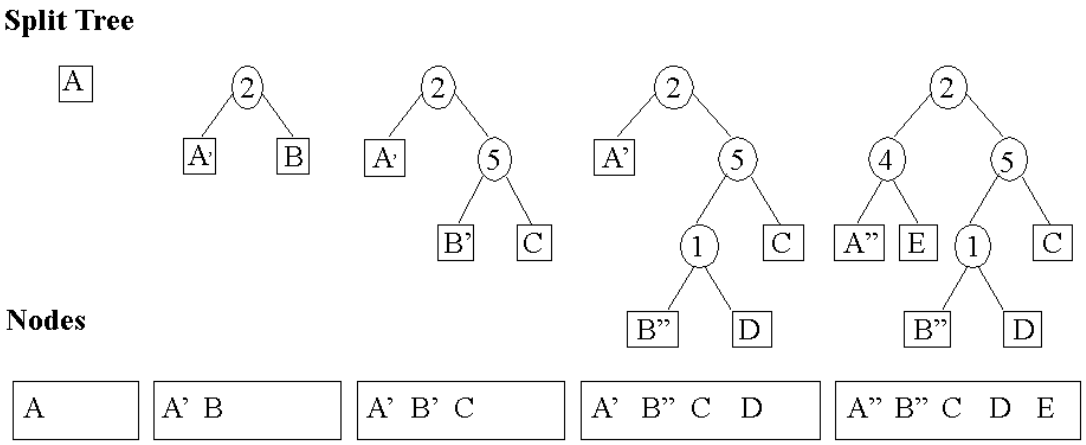


Figure 17: Example of a split history.

If it is now necessary to split the node containing A'', B'', C, D and E, we have to choose a split axis first. If we would choose dimension 1, for example, we had to put A'' and E into one of the partitions. However, those two nodes have never been split according to axis 1 and therefore span the whole data space along this axis. Consequently, the MBR of this partition will also span the whole data space. Regardless of the shape of the other partition, this leads to a high overlap and deteriorates query performance. Only if we choose dimension 2 as split axis, an overlap-free split can be determined, as every subnode has already been split according to this dimension. The X-tree utilizes this observation by always using the split dimension in the root node of the particular split tree. Obviously, it is possible that this results in an unbalanced split, which would create one underfilled and one almost overflowing node. As a result, the storage utilization would decrease and the directory would degenerate. To overcome this problem, the X-tree does not split, but creates an enlarged directory node instead – a supernode. Supernodes span multiple disk blocks and the probability of their creation increases with the dimension of the data space. To ensure efficient operation in lower-dimensional spaces, the X-tree includes a geometric split algorithm, too. Therefore, the X-tree split algorithm consists of at most three steps. First the R^*-tree split algorithm or any other topological split algorithm is applied. If this results in highly overlapping MBR's, the overlap-free split algorithm described above, is used. If this in turn leads to an unbalanced directory, a supernode is created. Figure 18 shows the entire split algorithm of the X-tree.

Apart from the split algorithm for directory pages, the X-tree uses the same insertion process as the R^*-tree. Especially the algorithm for choosing the appropriate subtree for the insertion of a new object is not altered.

In medium-dimensional spaces, the X-tree performs orders of magnitude better than the R^*-tree for all query types. For smaller dimensions the behavior is almost identical and in higher dimensions the X-tree, too, has to visit such a high number of data pages that a linear scan is less expensive. It is impossible to give exact numbers here, as a lot of factors influence the performance of an index structure.

```
bool XDirNodeSplit (MBRSet in, MBRSet out1, MBRSet out2)
{
    MBRSet t1, t2;
    MBR r1, r1;
    /* first try a toplogical split */
    TopologicalSplit(in, t1, t2);
    r1 = CalculateMBR(t1);
    r2 = CalculateMBR(t2);
    if (Overlap(r1, r2) > MAX_OVERLAP)
    {
        /* try overlap-minimal split */
        OverlapMinimalSplit(in, t1, t2);
        if ((t1.num_of_MBRs < MIN_FANOUT) ||
            (t2. num_of_MBRs < MIN_FANOUT)
            /* create supernode*/
            return FALSE;
    }
    out1 = t1;
    out2 = t2;
    return TRUE;
}
```

Figure 18: Split-Algorithm of the X-tree.

2.4 The SS-tree

The SS-tree [WJ 96] extends the R*-tree in a different direction than the X-tree. It no longer uses hyperrectangles as page regions, but hyperspheres. To achieve better efficiency, the spheres are not minimal bounding spheres. Instead, the centroid point, i.e. the average value in each dimension, is used as a center point and the minimum radius is chosen, such that all objects are included in the sphere. Therefore, the region description consists of the centroid point and the radius. This allows a very efficient determination of distances between page regions or query objects and page regions.

The insertion algorithm is similar to the one for the R*-tree, but is adapted in every major stage. The SS-tree uses the concept of a forced reinsert, too. Unlike in the R*-tree, every object that is inserted is in the first step added to a reinsertion list, which is emptied afterwards. During the descent in the index, the child node whose

centroid point is closest to the object is chosen for insertion. The volume of the page regions or the amount of overlap enlargement is not considered at all. Of course, the new centroid point and the new radius for each node on the insertion path must be determined on the way down. When an overflow condition occurs, again a forced reinsert operation is raised, where 30% of the objects with the highest variance from the centroid point are deleted and reinserted after the page regions were updated. If a split is necessary, the dimension with highest variance is chosen as split axis. By examining all possible split positions, which fulfill the space utilization threshold, the split plane is determined. The split plane with the minimal sum of variances on each side is chosen. Again the amount of overlap that is produced, is in no way optimized.

This leads us to a main problem of the SS-tree. The fact that spheres are used as page regions, makes it very hard to minimize the amount of overlap in the index. In fact, an overlap-free split can be impossible in some situations. For this reason, the SS-tree outperforms the R^*-tree by a factor of two, but it does not reach the performance of the X-tree.

2.5 The SR-tree

The SR-tree [KS 97] can be regarded as a combination of the R^*-tree and the SS-tree, as it uses the intersection between a hyperrectangle and hypersphere as page region. The rectangular part is the axis parallel minimal bounding rectangle of all objects, like in the R^*-tree. The spherical part is the minimum sphere around the centroid point of the stored objects, analogously to the SS-tree. This results in the most complex description for page regions of all index structures presented here. It consists of (2*d) floating point values for the MBR and (d+1) values for the sphere. An example of the page regions of the SR-tree is shown in figure 19.

According to Katyama and Satoh [KS 97], a study of the properties of spheres and rectangles provided the motivation for using this combination as page region. They state that spheres are better suited for nearest neighbor queries and range queries, if an L_2-metric is used. This is because spheres tend to have a smaller diameter than rectangles. On the other hand, rectangles have a much smaller volume than bounding spheres. On the average the former are about 2% of the latter in their experiments. This results in a much smaller overlap, whereas spheres usually overlap largely as

described above. The authors therefore believe that a combination of the two will overcome both problems.

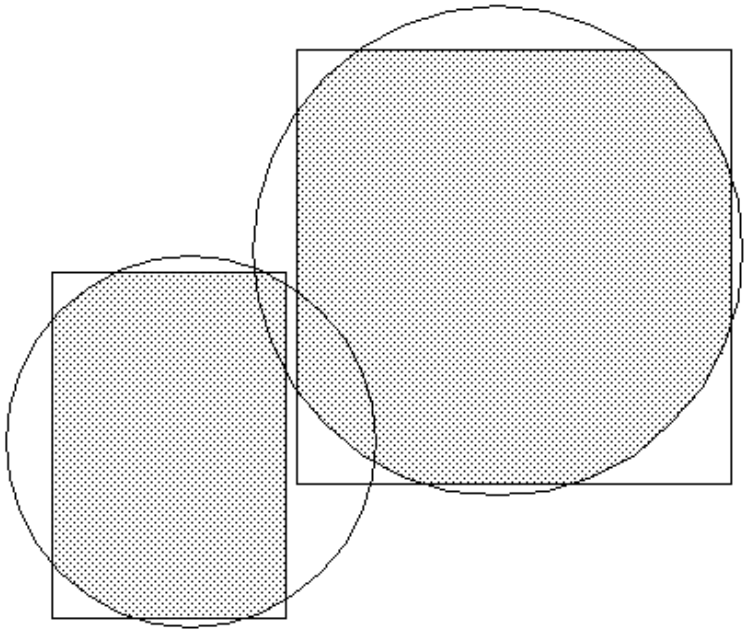


Figure 19: Page Regions of the SR-tree.

The insert and split algorithms are taken from the SS-tree and only slightly modified. Besides the necessity of updating the MBR after an insertion, the new bounding sphere is calculated utilizing both the bounding sphere and the bounding rectangle of the children, which results in smaller radius's. However, no information from the MBR's is used for the choice of the subtree or the determination of the split.

The SR-tree uses the following definition of the distance between a query point *q* and a region *R*:

Definition 6: Distance to the page region in the SR-tree

$$Dist(q, R) = \max(Dist(q, R.MBR), Dist(q, R.sphere))$$

Unfortunately, this is not the minimal distance to the intersection solid, as explained in section 3.7. The distance to the MBR as well as the distance to the sphere is smaller than the true distance to the intersection solid. However, it can be shown that the above function is a lower bound of the correct distance function. This guarantees

that the processing of neither range queries nor nearest neighbor queries produces false dismissals, but obviously some optimization potential is wasted in these cases.

The performance results presented in [KS 97] suggest that the SR-tree outperforms the R*-tree as well as the SS-tree. It still remains an open question whether it also outperforms the X-tree, as no experimental comparison has been made yet, to the authors best knowledge. Comparing the performance of the two structures to the performance of the R*-tree, one could conclude that the SR-tree does not reach the performance of the X-tree, but this results are rather uncertain.

Another aspect of the SR-tree, however, is certain. As it uses such a complex description of the page region, the fanout is massively decreased, which results in a higher index. Therefore, the SR-tree needs even more directory page accesses than the SS-tree.

2.6 The TV-tree

The TV-tree [LJF 95] was designed for real data with the Karhunen-Loève-Transform in mind. This mapping, also known as principal component analysis, preserves distances and eliminates linear correlation's. The resulting vectors have a high variance and therefore a good selectivity in the first few dimensions. The last few dimensions, however, are of minor importance for the query processing. This gives us two guiding principles for the use of Karhunen-Loève-transformed data. First, branching according to the first few attributes should be performed as early as possible, i.e. at the topmost levels of the index. This has the effect that the extension of the regions in lower levels of the tree is often zero in these dimensions. The second point is that it is not sensible to split the data space in the dimensions corresponding with the last few attributes, as those are never used for pruning branches during query processing.

Therefore, the TV-tree describes regions by using so-called Telescope Vectors (TV). These vectors can be dynamically shortened and are divided into α active and k inactive dimensions. The inactive dimensions form the greatest common prefix of all vectors stored in the subtree. Consequently, the page regions have an extension of zero in those dimensions. In the α active dimensions, however, the regions have the shape of an L_p-sphere, with p being either 1, 2 or ∞. The region is assumed to have infinite extension in the remaining dimensions, which are either active on lower

levels of the index or of minor importance for the query processing. Figure 20 shows a telescope vector and its main components.

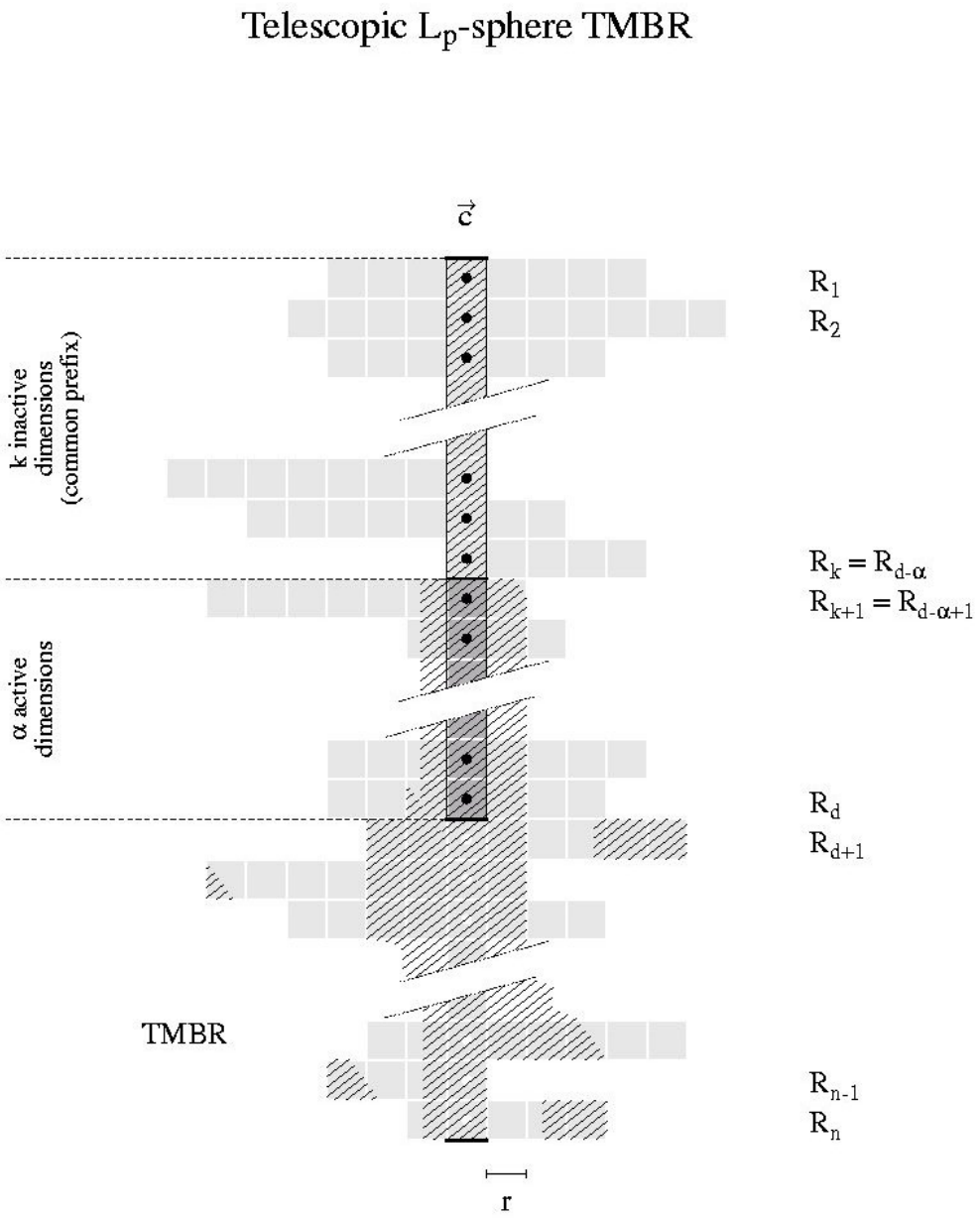


Figure 20: Structure of a Telescope Vector.

Consequently, the region description consists of α floating point values for the center point in the active dimensions and one float value for the radius. The values for the inactive dimensions are stored at the level, where a dimension turns from active to inactive, i.e. on higher levels of the tree. The number of active dimensions α is constant on all levels to achieve a uniform capacity of directory nodes. Experiments

show that a low number of active dimensions (α = 2) yields the best search performance.

The choice of the appropriate subtree during the insert operation is based on the following criteria (ordered by decreasing priority):

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

Again a reinsert operation, similar to the R*-tree, is performed prior to a split. The split starts with choosing two seed-points (or seed-regions in case of a directory page), which have the least common prefix. In tie situations, the maximum distance is used as an additional criterion. Afterwards, the objects are inserted in one of the new subtrees. The above criteria are used to determine the subtree, as long as the minimum storage utilization is guaranteed.

The concept of the telescope vectors was introduced to increase the capacity and therefore the fanout of directory pages. This makes it necessary that the data can be Karhunen-Loève-transformed, i.e. their dimension can be ordered by their significance. Also there must exist such feature vectors that allow the shift of active dimensions. If these preconditions are satisfied, the authors report a good speed-up compared to the R*-tree. Nevertheless, for uniform data and real data, which does not conform to the conditions stated above, experiments [BKK 96] show that the X-tree outperforms the TV-tree.

3 Regions

As stated above, data organizing index structures use page regions to partition the data space and to approximate spatial objects. The form and representation of these regions has great influence on the efficiency of an index structure. Therefore, we will discuss such region descriptions more thoroughly now. After some general remarks, a few of the common forms of page regions are discussed in detail.

3.1 The Purpose of Regions

The page regions of an index structure not only partition the data space. They also provide a possibility to exclude portions of the tree from further query processing. As all objects in a subtree are contained in the corresponding page region, subtrees whose page regions do not contain the query point can be excluded from query processing. Of course, this should be done on the lowest possible level of the tree, to achieve maximum query performance.

However, regions fulfill another purpose: they are used to approximate spatial objects. Spatial objects are often large in terms of memory needed to describe them and even simple operations, e.g. a test for equality of two objects, are not always trivial. This makes it necessary to approximate the objects in order to get a more compact representation, which is easier to handle. Besides this, a smaller object description allows it to store more objects in the nodes of a tree. This results in smaller trees and improved query performance.

As the page regions of the lower levels of the tree are in fact spatial objects, page regions are approximations even in index structures designed solely for point objects. Therefore the terms *page region* and *approximation* are used synonymously from now on.

3.2 Properties of Region Descriptions

To achieve these aims, the descriptions of page regions have to fulfill certain properties.

3.2.1 Approximations

In order to maintain the correctness of the query algorithms, the distance between two page regions must be a lower bound of the distance between the two closest objects contained in those regions. Otherwise false dismissals may occur, for example during the processing of a nearest neighbor query, because the pruning heuristics depend on this. An approximation of an object that fulfills this property is called *conservative*. On the other hand of course, the region should not contain too much of dead space, i.e. data space covered by the page region, but not by any object in the page region. Dead space does not affect the correctness of the query algorithms but their efficiency. It generally leads to more overlap and prevents the early exclusion of branches from the search process. Obviously, the optimization of the dead space a region description produces must be done very carefully, to maintain a conservative approximation.

As we have seen, the quality of an approximation depends on the amount of dead space covered by it. To make this independent from the object size the following definition is made.

Definition 7: Quality of an Approximation
The quality of an approximation A of the object O is defined as the volume of the object divided by the volume of the approximation.

$$Quality(A) = \frac{volume(O)}{volume(A)}$$

Obviously, this definition does assume approximations of spatial objects with a volume greater than zero and is therefore only sensible on directory levels. The necessity of a conservative approximation remains independent from this definition.

The use of approximations instead of the real objects produces some problems that have to be addressed. One point is that if spatial objects are stored in the database, a multi-step query processing must be used. As two different objects may have the

same approximation, they are both reported as answers to the query. A second *refinement step* has to be performed after the query was processed. The first step is therefore called *filter step*. Figure 21 shows the entire query process.

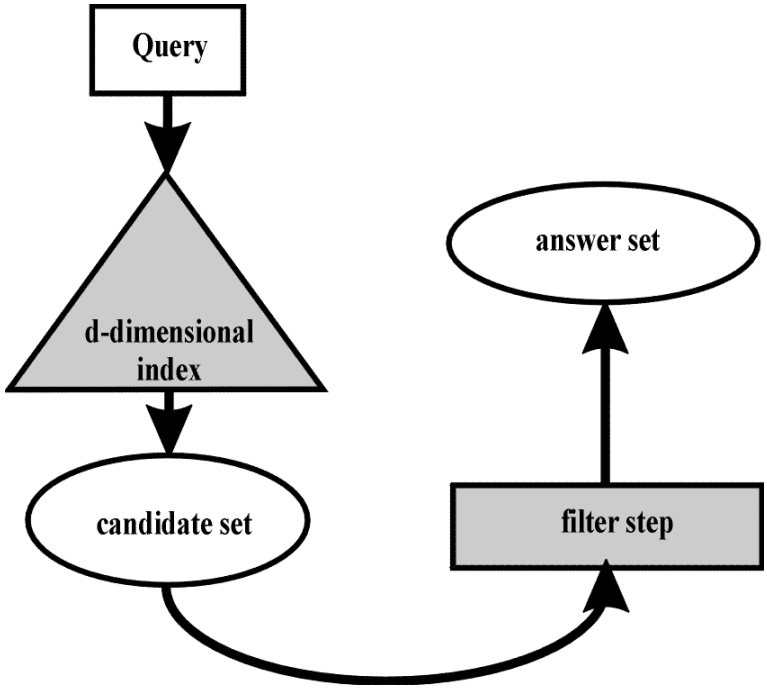


Figure 21: Multi-Step Query Process.

Of course, the complete query processing can only be efficient if a substantial number of objects are excluded already in the filter step. Otherwise, the overall processing time would be increased by the use of approximations. The meaning of the word *substantial* in this context depends on the relation between the time needed to process an approximation and the time needed for a real object. Obviously, a better approximation usually allows the exclusion of more objects. The quality of the approximation should therefore be maximized.

This leads to another problem in this context. Approximations of a better quality usually have a more complicated description. Consequently, more memory is needed to store one approximation, which in turn decreases the number of objects that can be stored in one node. This increases the height of the index and, therefore, negatively affects the query performance. This is contradictory to the reason why approximations were introduced in the first place. Therefore, a balance between the

quality of the approximation and the memory needed to store its description must be found.

3.2.2 Complexity of the operations

Another important point is the complexity of the operations that are needed for the query processing. As we have seen in the previous chapter, a few operations have to be defined for query regions in order to process the query types we identified earlier. These are namely:

- A distance function between page regions and between page regions and query objects.
- A test if an object is contained in a page region.
- A test if two page regions intersect.

In addition to this, functions depending on the specific index structure have to be provided, e.g. the amount of overlap has to be determined during the insertion process of the R^*-tree and the X-tree. As all of these operations have to be performed frequently during query processing, they should all be easy to evaluate for maximum query performance. This is especially important for high-dimensional data spaces, as the complexity of these operations is usually a function of the number of dimensions.

3.3 Minimal Bounding Rectangles

Minimal bounding rectangles (*MBR's*) are defined by the maximal extension of the contained objects in each dimension. MBR's can either be axis parallel or of arbitrary orientation (often called *rotated bounding rectangles*). For the latter, the direction of the rectangle has to be stored, too. The additional degree of freedom allows an approximation of better quality, but needs more storage space (at least 2*d+3 floating point values) and increases the complexity of the operations. Consequently, Ablaßmeier found out that some basic operations take at least four times as long for rotated bounding rectangles than for axis parallel ones [Abl 93]. Therefore, usually only axis parallel MBR's are used, like in the R^*-tree and the X-tree. The term *MBR* will therefore stand for *axis parallel minimal bounding rectangle* from now on.

For the encoding of a MBR 2*d (with d denoting the number of dimensions of the data space) floating-point values are necessary. The determination of the distance to a MBR is rather easy. The distance is calculated for each dimension separately and summed up afterwards. Figure 22 shows the algorithm for the L_2-metric. The tests for containment and intersection are reduced to the test if the distance to the MBR is zero or not. Even the calculation of the overlap is just a small extension of this algorithm.

```
float DistToMBR(Point p, MBR m)
{
    int i;
    float d; /* distance in the current dimension */
    float SquareSum; /* sum of the squares of d */
    for (i = 0; i < dimension; i++)
    {
        if (p.coordinate[i] < m.min[i])
            d = m.min[i] – p.coordinate[i];
        else if (p.coordinate[i] > m.max[i])
            d = p.coordinate[i] – m.max[i];
        else d = 0;
        SquareSum = SquareSum + d * d;
    }
    return sqrt(SquareSum);
}
```

Figure 22: Euclidean Distance to a MBR.

Compared to bounding spheres, MBRs have smaller volumes on the average, which again helps to reduce the overlap. A side effect of the definition of a MBR is that each hyperplane marking the boundary of the MBR contains at least one point of an object in this page region. This makes it easy to determine the maximal distance of the closest point inside the MBR to a query point. The RKV algorithm for nearest neighbor queries takes advantage of this fact. Generally, this supports the efficiency of any nearest neighbor algorithm.

MBR's are used in for example in the R^*-tree and the X-tree.

3.4 Polygons

Rectangles are quite well suited for page regions. The idea to further optimize them by introducing more corners seems obvious. The resulting polygons should have less

volume and the quality of the approximation would therefore increase. Unfortunately, this advantage is outweighed by a number of disadvantages the use polygons brings with it.

First, the corners of the polygon must be stored. Depending on the number of corners that are used, the amount of memory needed would be very large. For the convex hull, the number of corners can not even be told in advance. In addition to this, it may change from object to object. Nevertheless, no polygon can be described with as little as 2*d floating point values, like a MBR. Therefore, the use of polygons would result in a smaller capacity of the nodes and in higher indexes, with the known decrease in performance.

The use of polygons also makes the basic algorithms more difficult. The determination of a volume-optimized polygon is not as easy as finding the maximal extension of the page region. On the other hand, giving up the goal of optimizing for volume would also mean sacrificing the only advantage of using polygons. The calculation of the distance to a polygon requires the determination of the vertex that is closest to the query object. This can only be done by examining all corners of the polygon. To determine the volume of the overlapping region of two polygons, the resulting polygon has to be constructed and its volume must be computed. The computation of a polygons volume in itself is far from trivial.

The performance of an index structure using polygons as page regions should therefore not be as good as with MBR’s a page regions. This might be the reason that, to the authors best knowledge, there exists no high-dimensional index structure using polygons as page regions.

3.5 Spheres

Another very common form for page regions is the sphere. A sphere is completely described by its center point and the radius, which makes only d+1 floating point values necessary for the storage of the region. This makes it the most compact description of all those presented in this chapter. Therefore, index structures using spheres as page regions generally have the greatest node capacity of all.

Spheres have a small margin compared to their volume. Their high volume, especially in higher dimensions, is one reason why index structures based on spheres tend to suffer from a high overlap. Another reason for this is that an overlap-free split

is often not possible (cf. figure 23). Obviously, spheres are better suited for more compact roundish data objects. If the objects that have to be approximated are rather long and narrow, the bounding spheres will cover a lot of dead space, which again has a negative affect on the query performance.

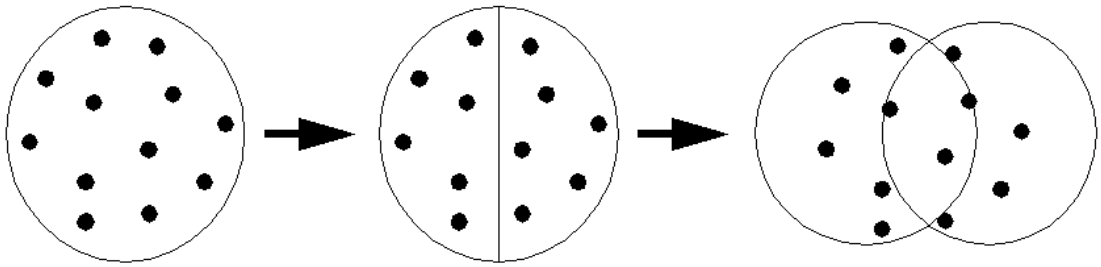


Figure 23: Situation where an Overlap-Free Split is impossible with Spheres.

The determination of distances and the tests for containment and intersection are very easy and straightforward. Usually the center point of the sphere is chosen to be the centroid of the objects to be contained. The coordinates x_i of the centroid x (x_1, ..., x_i, ..., x_d) are calculated as follows:

$$x_i = \frac{\sum_{k=1}^{n} C_k.x_i * C_k.w}{\sum_{k=1}^{n} C_k.w}$$

where k is an index to the children of the node, i is an index to the dimensions, $C_k.x_i$ denotes the i-th coordinate of the center of the k-th child and $C_k.w$ denotes the number of points contained in the subtree whose top is the child C_k. Both the SS-tree and the SR-tree use this definition. Unfortunately, this only guarantees for an object in the hemisphere directed to a query point, as depicted in figure 24. Therefore, the minimal distance to a sphere is less expressive than the minimal distance to a MBR for nearest neighbor queries. The performance for this query type can be affected negatively by this.

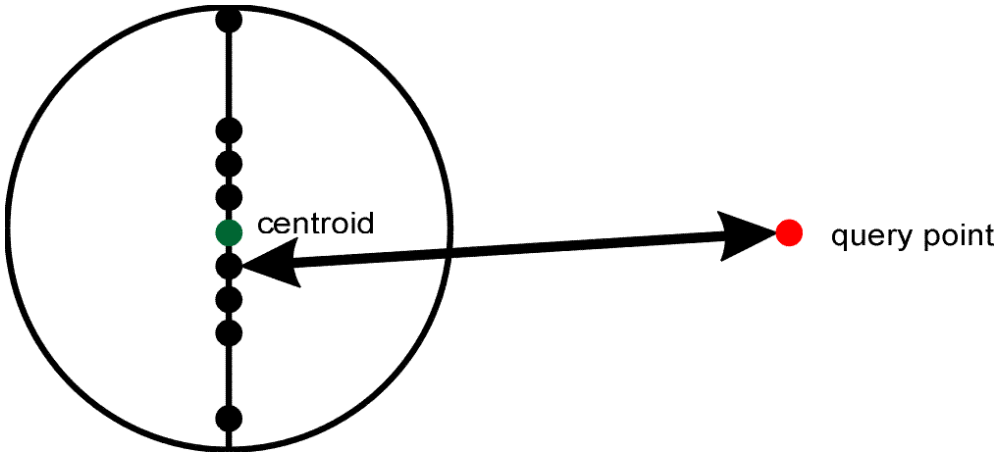


Figure 24: Closest Point in a Sphere (Worst Case).

The overlapping volume of two spheres is comparatively hard to compute, as it has a ‘egg-like’ shape. This might be one reason why neither the SS-tree nor the SR-tree optimizes for the amount of overlap an insertion of an object produces.

The SS-tree uses spheres as page regions, whereas the SR-tree uses them only as part of a more complex model for page regions (cf. section 3.7).

3.6 Ellipsoids

For objects that have a longer and narrower shape, an ellipsoid seems to be an appropriate approximation. If axis parallel ellipsoids are used, only d+2 floating-point values are needed to store the page region. Otherwise the directions of the two half-axis’ have to be stored, too.

A big problem with ellipsoids is the fact that all basic operations are rather complicated to perform. Even the construction of the ellipsoid bears many problems in it. As we have seen, an approximation should be optimized to have the least volume. However, another degree of freedom is introduced with the second radius, which complicates the optimization problem. The determination of distances and overlapping volumes also result in difficult optimization problems, which can not be solved efficiently.

Therefore, ellipsoids seem inappropriate as page regions and to the authors best knowledge there exists no index structure that uses them.

3.7 Intersection of Sphere and Rectangle

The SR-tree uses a different method to improve the form of its page regions. It tries to combine the advantages of rectangles and spheres by using the intersection of these two figures. To store this page region, the MBR and the centroid plus its radius are stored. This makes a total of 3*d+1 floating point values necessary, which makes it the largest description of a page region of all index structures presented in this thesis. The experimental results in [KS 97] show that this already leads to more directory node accesses than an SS-tree or R^*-tree need. This is still outweighed by the substantial decrease in data node accesses. Nevertheless, it suggests that a even larger description of a page region would result in a too high index, in the sense that the performance gain of the better approximation is more than outweighed by the performance loss due to the index height.

The basic algorithms for this kind of page regions are comparatively easy, as they are the concatenation of the algorithms for rectangles and spheres. To test if a query point is inside the region, it is tested if it is inside the sphere and inside the rectangle. If both are true, the point is also inside the intersection. Although this test is more complex than the single tests, its complexity is still of the same degree *O(d)* (with d denoting the number of dimensions) as the single operations. The test for intersection is again reduced to the question if the distance is zero or not.

The distance function proposed by Katayama and Satoh, however, is incorrect in some constellations. As mentioned in the previous chapter, they define the distance between a query point *p* and a page region *R* as follows:

$$Dist(q, R) = \max(Dist(q, R.MBR), Dist(q, R.sphere))$$

Unfortunately, this is not correct for some query points, as depicted in figure 25. For all query points P in the area A the correct distance would be the one from P to M_T, but the algorithm chooses either the one to M_S or M_R. Obviously, these distances are both lower boundaries of the true distance, which ensures that no false dismissals occur during the processing of range or nearest neighbor queries. The max-function in the definition even ensures that the error is minimized. The error that is made can even be calculated, as the following lemma shows.

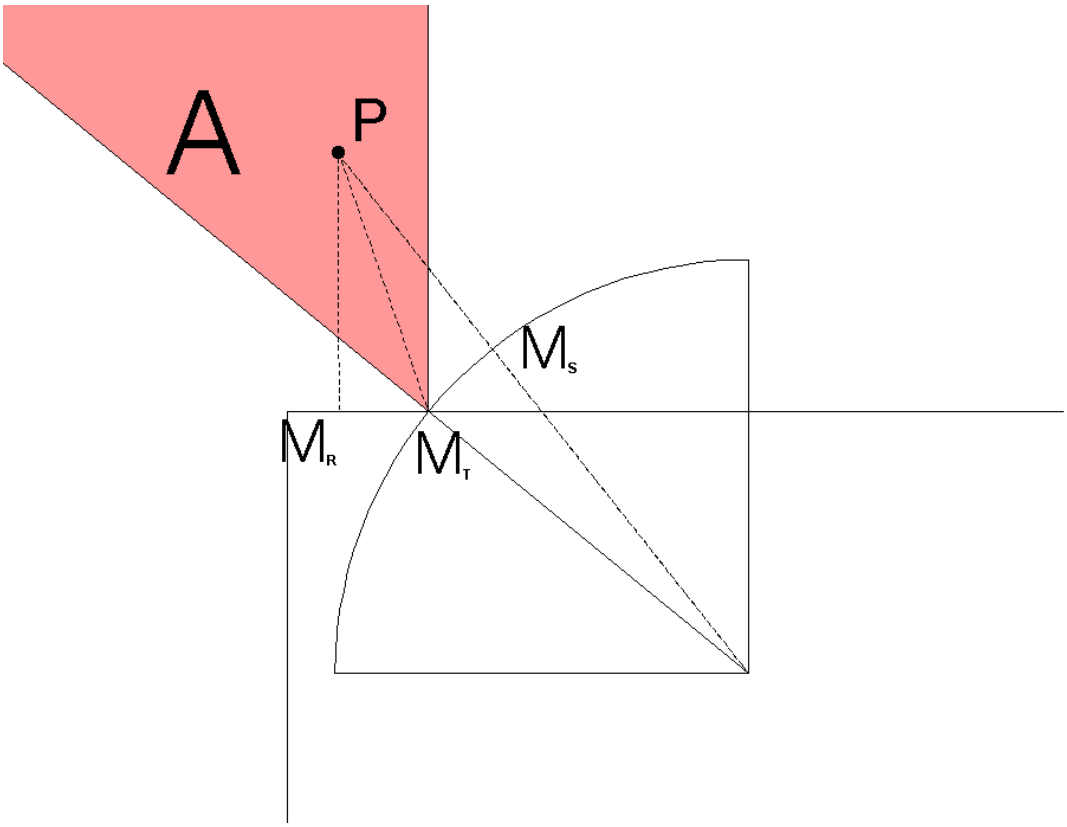


Figure 25: Incorrect Euclidean Distance in the SR-tree.

Lemma 1: Relative Error of the SR-tree Distance Function

The relative error of the SR-tree distance function is:

$$\frac{\overline{PM_T} - \overline{PM_X}}{\overline{PM_T}} = 1 - \frac{\overline{PM_X}}{\sqrt{\overline{PM_R}^2 + \overline{M_R M_T}^2}} \quad with\, M_X \in \{M_R, M_S\}$$

Proof:

Without loss of generality, we assume the situation depicted in figure 25. The relative error of the SR-tree distance function can then be calculated as follows:

$$\overline{PM_T} = \sqrt{\overline{PM_R}^2 + \overline{M_R M_T}^2}$$

$$\Rightarrow \quad \frac{\overline{PM_T} - \overline{PM_X}}{\overline{PM_T}} = \frac{\sqrt{\overline{PM_R}^2 + \overline{M_R M_T}^2} - \overline{PM_X}}{\sqrt{\overline{PM_R}^2 + \overline{M_R M_T}^2}}$$

$$\Rightarrow \quad \frac{\overline{PM_T} - \overline{PM_X}}{\overline{PM_T}} = 1 - \frac{\overline{PM_X}}{\sqrt{\overline{PM_R}^2 + \overline{M_R M_T}^2}} \quad with\, M_X \in \{M_R, M_S\}$$

□

As we have seen the use of this distance function does not endanger the correctness of the queries algorithms, but some optimization potential is wasted. Still the experimental results in [KS 97] show that this approximation is an improvement over the use of rectangles or spheres solely.

3.8 Summary

In this chapter, we have seen that page regions are used to group the objects in a node, to prune branches during query processing and to approximate spatial objects. As page regions themselves are spatial objects, their grouping on higher levels of the index is an approximation, too. The terms page region and approximation can therefore be used synonymously in this context.

MBR’s and spheres are most common types of shapes for page regions in index structures. Nevertheless, they have particular disadvantages, which can be overcome by combining these two concepts.

4 The XO-Tree

Based on the facts presented in the previous two chapters, we will now introduce a new index structure for high-dimensional data spaces: The XO-tree. After clearly defining our goals, the structure of the XO-tree will be presented. In particular, the new type of page region that this index structure uses will be thoroughly discussed. In the following section, the algorithms used by the XO-tree will be explained.

4.1 Design Objectives

We already recognized that special index structures are needed for high-dimensional data spaces. The main problems in this area are the high overlap in the directory of the index and the size of the data objects in terms of memory that is needed to store them. To minimize these problems there are two parts of an index structure where special care has to be taken during the design process. These two parts are the insertion process and the choice of the page region that is used.

4.1.1 The Insertion Process

The insertion process is extremely important for the query performance of an index structure. To achieve a well-balanced index with as little overlap as possible, there are again two major points that have to be examined.

First, the decision into which subtree an object is inserted has to be made very carefully. Although forced reinserts can make up for some of the mistakes made during this phase, a good choice of the subtree is essential for the quality of the index. As Berchtold, Keim and Kriegel showed in [BKK 96], the heuristics that are applied should optimize the amount of overlap, the insertion of the object produces. This seems to be the most important point for the optimal choice of the subtree.

Properties like the volume of the resulting page region or its margin, on the other hand, are of minor importance.

The split heuristic is the other significant characteristic of the insertion process. The result of a split should be well balanced to prevent a deteriorated index. On the other hand, a well balanced but highly overlapping split result is equally undesirable. Therefore, if an overlap-minimal split results in an unbalanced situation, a split should be avoided at all. In these cases, a linear scan of the associated subtree is more efficient. Therefore, an index structure should be as hierarchical as possible and as linear as necessary. This can be achieved by the application of the supernode concept, as introduce with the X-tree.

4.1.2 The Page Region

Finally, the form of the page region has an influence on the performance of an index structure. In the previous chapter, we saw that a balance between the quality of the approximation and the complexity of its description has to be found. The results of the experiments with the SR-tree show that a considerable performance gain can be achieved by the use of a combination of sphere and rectangle. The main advantage of this form of the page region is the reduction of dead space. Especially the corners of the MBR are often cut off by the bounding sphere. This effect gets even more important in higher dimensions, as the number of corners of a MBR grows exponentially with the dimension. However, the great size of the description uses up some of the performance gained by the increased quality of the approximation. This shows in the high number of directory node accesses that the SR-tree makes compared to the SS-tree. Obviously, the goal must be, to reduce the dead space and at the same time increase the size of the region description as little as possible.

4.2 The Page Region of the XO-tree

To achieve the goals stated in the previous section, the XO-tree uses new forms of page regions. The common objective is to cut off the corners of a rectangle as far as they cover only dead space. Obviously, there are several ways, how to cut off corners of a rectangle, which contain no objects. One would be the combination of rectangle and sphere as used by the SR-tree. But as we have seen above, this results in higher indexes, because the storage of the approximation needs a lot of memory. To avoid

this unwanted effect, three different approaches were implemented and tested for the XO-tree. They will be described in detail now. If nothing else is mentioned, the letter *d* denotes the number of the dimensions of the data space in this chapter.

4.2.1 Ovaloids

At first, we will observe ovaloids. Ovaloids can be viewed as rectangles with round corners. They are always associated with a “inner” rectangle, which is the rectangle mentioned in the following definition:

Definition 8: Ovaloid

The ovaloid $O_{R,r}$ that is associated with the rectangle *R*, is the geometric location of all points that have an Euclidean distance of at most *r* from *R*.

Figure 26 shows an ovaloid in two-dimensional space. As one can see from the definition, there are not only the corners excluded from the rectangle. In three and higher dimensions even entire edges, i.e. (d-2)-dimensional objects, are excluded from the rectangle. This further reduces the dead space that is covered by the approximation. On the other hand, the radius of the ovaloid is zero, if an object that has to be enclosed, resides on such an edge. In these cases, the ovaloid is identical with a MBR of the data.

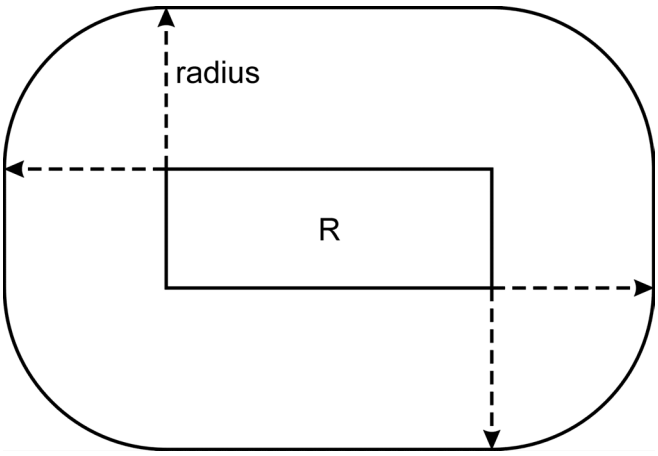


Figure 26: Two-Dimensional Ovaloid.

Obviously, there are 2*d+1 floating point values needed to store an ovaloid. This makes its description almost as small as the one for a rectangle. Consequently, nearly

the same number of entries as in an X-tree node, can be stored in the nodes of an index that uses ovaloids as page regions. This is considerably more than in the nodes of a SR-tree.

The basic operations can be evaluated rather easy, apart from the exact calculation of the overlap between two ovaloids. The distance between an object and an ovaloid can be defined utilizing the distance between the associated rectangle and the object:

Definition 9: Distance between an object and an ovaloid

The distance between an object O and an ovaloid $OV_{R,r}$ is defined as follows:

$$dist(O, OV_{R,r}) = dist(O, R) - r$$

The distance between two ovaloids is determined analogously, by calculating the distance between the two associated rectangles and subtracting the two radiuses.

There are no special cases where these functions do not yield the correct Euclidean distance to the page region, as there are for the page region of the SR-tree. This is an obvious advantage of an ovaloid over the intersection between rectangle and sphere.

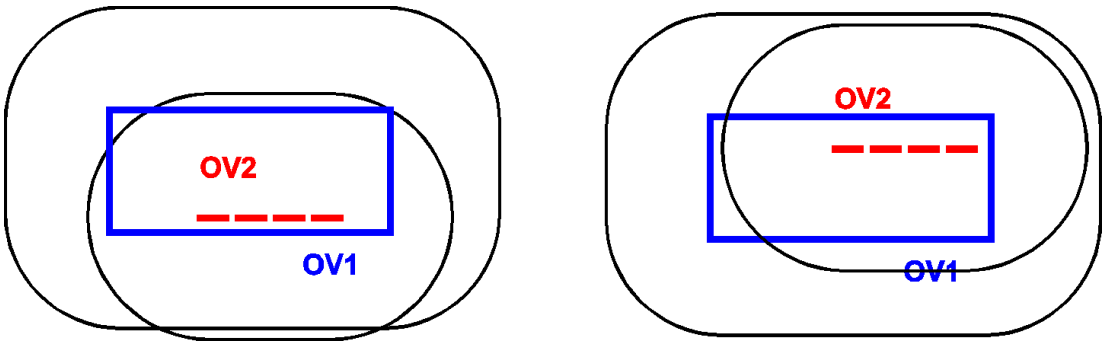


Figure 27: Test for Containment of an Ovaloid.

To test if a point is inside an ovaloid, the distance to the associated rectangle is calculated. If this distance is less than the radius, the point resides inside the ovaloid. The test for containment of another ovaloid is bit more complicated. Lets us assume that we want to test if ovaloid $O_{R1,r1}$ is fully contained in ovaloid $O_{R2,r2}$. For this, the maximal minimal distance between the two associated rectangles R1 and R2 has to be calculated. To this value, r1 has to be added. If the result is less than r2, $O_{R1,r1}$ is completely enclosed by $O_{R2,r2}$. Figure 27 shows the situation for 2-dimensional

ovaloids, whereas figure 28 depicts the pseudocode for the calculation of the maximal minimal distance between two MBR's.

Only the overlap of two ovaloids is hard to calculate. The objects that are formed from the intersection of two ovaloids are very complex. Nevertheless, there is an obvious upper boundary for this value. It is the overlap of the MBRs of the two ovaloids. These MBRs can easily be determined by adding the radius of the ovaloid to the absolute value of the coordinates of the inner rectangle. This way an upper boundary of the true overlap can be determined efficiently. The difference between this value and the true amount of overlap can be ignored. The use of another page region, like the one of the SR-tree, would result in a similar error, as the calculation of the true overlap would not be less difficult.

```
float MaximalDistanceOfMBRs(Ovaloid O1, Ovaloid O2);
{
    /* calculate the squared maximal distance of the associated MBRs of
    O1 and O2 */
    int k;
    /* total squared distance and distance in current dimension */
    float square_dist = 0.0, d;
    for (k = 0; k < dimension; k++)
    {
        if (((O1.max[k] + O1.radius) > (O2.max[k] + O2.radius)) ||
           ((O1.min[k] – O1.radius) < (O2.min[k] – O2.radius) ))
            return false;
        if (O1.max[k] <= O2.max[k])
            if (O1.min[k] < O2.min[k])
                d = O2.min[k] – O1.min[k];
            else
                d = 0.0;
        else
            if (O1.min[k] > O1.min[k])
                d = O1.max[k] – O2.max[k];
            else if (abs(O1.max[k] – O2.max[k]) >
                    abs(O2.min[k] - O1.min[k]))
                    d = O1.max[k] – O2.max[k];
                else
                    d = O2.min[k] - O1.min[k];
        square_dist = square_dist + d * d;
    }
    return square_dist;
}
```

Figure 28: Maximal minimal Distance of two MBR's.

In summary, one can say that the use of ovaloids permits a compact description of a page region with little dead space covered by it. The basic operations can easily be implemented and evaluated efficiently. Therefore, ovaloids fulfill the requirements for page regions, which we defined in section 4.1.2.

4.2.2 Intersection between Ovaloid and MBR

Another promising approach is to use the intersection between the MBR of the data and an ovaloid as page region. This technique also allows reducing the dead space covered by the corners of the MBR. However, as we already saw with the SR-tree, storing two independent approximations is undesirable, because of the high complexity in space. The combination of an ovaloid and a MBR has advantages here. As each ovaloid is associated with a rectangle anyway, it is straightforward to use a rectangle that can easily be derived from the MBR as basis for the ovaloid. This way the space complexity of the approximation can be reduced significantly. The use of the MBR itself would be senseless for obvious reasons.

We use the following approach to derive the basic rectangle for the ovaloid from the MBR. First, the dimension d_{se}, in which the MBR has the smallest extension, is determined. Half of this extension is then subtracted from the absolute coordinates of the points defining the MBR. The resulting rectangle, which has zero extension in dimension d_{se}, is used as the basis for the ovaloid. The radius of the ovaloid is chosen appropriately to enclose all objects in the MBR. Again, the shape of this approximation is in extreme cases identical with that of the MBR. In the optimal case, the radius equals half the extension in dimension d_{se}. An even smaller radius is not possible, as not all of the (d-1)-dimensional hyperrectangles defining the MBR would be touched by the ovaloid. However, as we have seen in chapter 3.3, each of these hyperrectangles contains at least one data point.

In order to permit efficient access to the ovaloid, the dimension with the smallest extension is also stored. Therefore, (2*d+1) floating-point values plus one integer value are needed to store this type of page region. Although this is more than for a single ovaloid, there is still a considerable advantage over the approximation used by the SR-tree.

All the basic operations are built on the operations on the single rectangles and ovaloids. This makes the implementation and evaluation simple and still efficient.

The test for containment of an object is the straightforward combination of the two basic tests. Obviously, the same problem with the evaluation of the overlap as with the exclusive use of ovaloids comes up here, too. It is again solved by using the overlap of the MBRs as an upper boundary for the true value, but here the MBR does not have to be calculated, as it is already stored. This simple building process for the basic operations leads to a similar distance function like the one for the SR-tree:

Definition 10: Distance to the page region for the intersection of MBR and ovaloid

The distance between a point *q* and a page region *R* is defined as:

$$Dist(q, R) = \max(Dist(q, R.MBR), Dist(q, R.ovaloid))$$

This distance function obviously has the same properties as the one used with the SR-tree. This includes the erroneous distance value that is calculated in certain situations. Fully aware of all the disadvantages this type of page region posses, it was implemented and tested as part of the XO-tree, to evaluate the effect of these factors on the performance of the index structure.

4.2.3 The "Corner-cut" Approximation

We developed another type of page region that reduces the dead space covered by the corners of the MBR for the data. We call it the *corner-cut approximation* as it looks like an MBR whose corners were cut off with a knife. It is based on a MBR like the ovaloid. However, here the MBR is not inside the page region it is associated with, but it surrounds the page region. Apart from the MBR, the corner-cut approximation uses one additional floating-point value *r*. The approximation is then constructed from the MBR by the following process:

In each corner of a *d*-dimensional MBR, d points are created by subtracting *r* from the absolute value of the *i*-th ($i \in \{0,\ldots,d-1\}$) coordinate of the corner. All other coordinates are the same as the ones of the corresponding corner. These new points replace the corner from which they were created. Obviously, these points define a (*d*-1)-dimensional cutting plane between the MBR and the corner-cut approximation. Figures 29 and 30 show some examples for two and three dimensions.

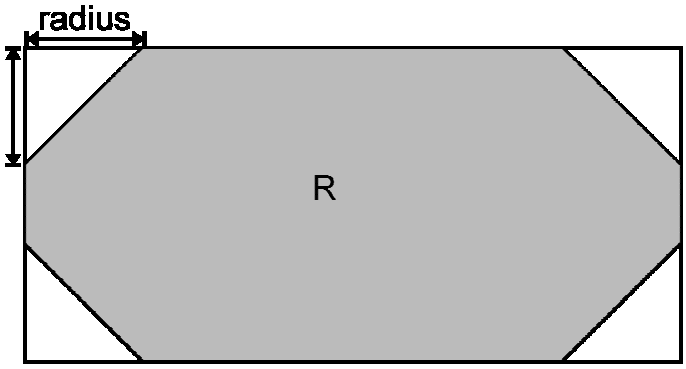


Figure 29: Corner-Cut Approximation (2D).

Of course, the value *r* has to be chosen appropriately so that all objects enclosed by the MBR are also enclosed by the corner-cut approximation. In the worst case this means that *r* has to be zero and the corner-cut approximation is then identical with the MBR. To store a corner-cut approximation, (2*d+1) floating-point values are needed. This is exactly the same as for an ovaloid is needed.

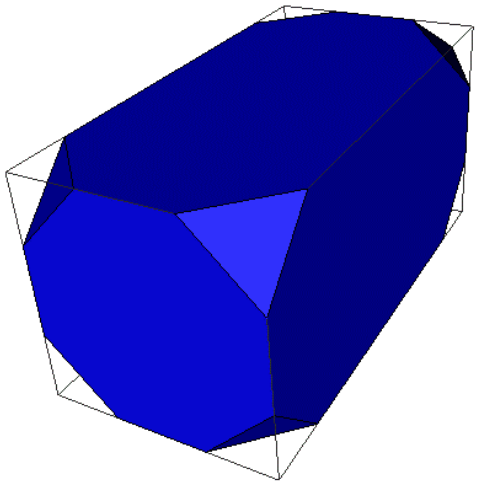


Figure 30: Corner-Cut Approximation (3D).

The reason why the corner-cut approximation was introduced, was that this type of approximation is very robust concerning rounding errors. The other two approximation types, on the other hand, lost some quality if a page region had to be adapted several times. As several tests showed, the reason for this effect is cumulating rounding errors in the calculation process.

Let us now examine the basic operations for the corner-cut approximation. There exists an easy and efficient method to determine whether a point is inside the page region. Two things are important for this. First, if the point is outside the MBR, it is also outside the corner-cut approximation. If this is not the case, then the point can only be outside the page region if its L_1-distance to its closest corner is less than the "radius" *r*. This test can be performed by examining all of the coordinates of the point once. Therefore, it has a worst case time complexity of *O(d)* with *d* again denoting the number of dimensions of the data space. The pseudocode for the algorithm is presented in figure 31. To determine if a page region is completely enclosed in the another one, an analogous test is performed which has the same complexity.

```
bool contains (Point p, Corner_Cut cca)
{
    int   k; // index for the loop
    float L1_dist = 0.0; /* L1-distance of the point to the corner */
    for (k = 0; k < p.dimension; k++)
    {
        if ((p.coord[k] > cca.max[k]) || (p.coord[k] < cca.min[k]))
        {
            return false;
        }
        if (p.coord[k] – cca.min[k] < (cca.max[k] – cca.min[k]) / 2)
            L1_dist = L1_dist + p.coord[k] - cca.min[k];
        else
            L1_dist = L1_dist + cca.max[k] – p.coord[k];
    }
    if (L1_dist >= cca.r)
        return true;
    else
        return false;
}
```

Figure 31: Test, if a point is inside a corner-cut approximation.

As with ovaloids, the calculation of the overlap of two page regions is difficult due to the many different and complex objects that can result from the intersection of those page regions. Consequently, we use the overlap of the MBRs of those page regions as an upper boundary for the true value. As the MBRs are stored as part of the corner-cut approximation, they can be accessed very efficiently.

To determine the Euclidean distance of a point to a corner-cut approximation the distance to the corresponding MBR is calculated. In certain situations this does not yield the correct distance value. The calculation of the correct distance in these situations would be very time consuming, as one would have to drop a perpendicular on a (d-1)-dimensional hypersurface. To avoid this costly operation, the distance to the MBR is used, which represents a lower boundary for the true value. This way, no false dismissals can occur during nearest-neighbor queries.

4.3 The Structure of the XO-tree

The structure of the XO-tree is similar to that of the X-tree. Especially the concept of supernodes and the overlap-minimal split are inherited from the X-tree. The data structures mainly differ in two aspects. First, of course, the XO-tree uses a different page region. The second point is that it allows supernodes on the level of data nodes, too. This is not the case for the X-tree. Obviously, this is only of importance for spatial data objects, as an overlap-free split is always possible for point data.

A data node contains entries consisting of a pointer to an actual data object and of its approximation. The entries in a directory node consist of a pointer to a subnode, its approximation and splithistory. The splithistory is implemented as a bitfield with d entries. In addition to this, every node stores the number of logical disk blocks it occupies. If a node uses more than one logical block, it is called a *supernode*. Except for the root node, every node is at least filled to some predefined degree. The capacity for one block of a data or directory node differs due to the different size of their entries. The maximal number of entries in a node depends on the number of blocks, this node occupies, but is always a multiple of the basic block capacity.

4.4 Algorithms of the XO-tree

We will now examine the dynamic features of the XO-tree. The insertion process will be thoroughly discussed, followed by some considerations about the deletion of objects. A look at the query algorithms of the XO-tree closes this section.

4.4.1 Insert

As we mentioned in section 4.1.1, the insertion algorithm should be optimized to produce as little overlap as possible. The XO-tree's insertion algorithm is therefore based on the algorithm of the X-tree. Nevertheless, is has undergone significant changes.

```
PageAdr Insert(Point object, PageAdr pa)
{
    Page p = LoadPage(pa);
    PageAdr subtree;
    PageAdr new_son;
    PageAdr brother = NULL;
    if (IsDataPage(p))
        InsertObjectInPage(object, p);
        if (Overflow(p))
            brother = OverflowTreatment(p);
    if (IsDirectoryPage(p))
        subtree = ChooseSubtree(p, object);
        new_son = Insert(object, subtree);
        if (new_son)
            InsertSonInPage(new_son, p);
            if (Overflow(p))
                brother = OverflowTreatment(p);
    return brother;
}
```

Figure 32: Insert-Algorithm of the XO-tree.

The basic structure is still the same. Starting with the root node, an appropriate subtree is chosen into which the object has to be inserted. This is repeated until a data node is reached. The object is then inserted into this node. If this yields an overflow, the node is either split or a supernode in created or extended. No reinsert mechanism is integrated as it is the case for the R^*-tree and SR-tree. The pseudocode for this algorithm is shown in figure 32.

The first change to the insertion process takes place in the algorithm for the choice of the subtree. As we mentioned in chapter 2, the X-tree uses the same algorithm as the R^*-tree for this task. This algorithm, which is shown in figure 13, has different optimization goals for data and directory nodes, if the object does not reside in any page region so that one has to be adapted. If the subtrees of the current node are data nodes, the subtree with the smallest enlargement in overlap is chosen. In cases of ambiguity, the enlargement in volume and the volume itself are further criteria. If, on the other hand, the subnodes are directory nodes, a different strategy is applied. Here the one, which yields the least enlargement in volume and in case of doubt, the one with the smallest volume, is chosen. This distinction between the two node types provides a slightly better performance for 2D rectangles, as stated in [BKSS 90].

```
PageAdr ChooseSubtree(Page p, Point object)
{
    int i;
    PageAdr subtree;
    float MINOverlEnl = INFINITY, MINVolEnl = INFINITY;
    float MINVol = INFINITY;
    float OverlEnl, VolEnl, Vol;
    /* Determine minimum overlap enlargement! */
    for (i = 0; i < p.num_objects; i++)
    {
        OverlEnl = OverlapEnlarge(p.childnode[i], object);
        VolEnl = VolumeEnlarge(p.childnode[i], object);
        Vol = Volume(p.childnode[i]);
        if ((OverlEnl < MINOverlEnl) || (OverlEnl == MINOverlEnl &&
        VolEnl < MINVolEnl) || (OverlEnl == MINOverlEnl &&
        VolEnl == MINVolEnl && Vol < MinVol))
        {
            MINOverlEnl = OverlapEnlarge(p.childnode[i], object);
            MINVolEnl = VolumeEnlarge(p.childnode[i], object);
            MINVol = Volume(p.childnode[i]);
            subtree = p.childnode[i];
        }
    }
    return subtree;
}
```

Figure 33: ChooseSubtree-Algorithm for the XO-tree.

Nevertheless, this obviously disagrees with the goals mentioned in section 4.1. As we want to minimize the overlap in the directory, this should be reflected in the

algorithm for the choice of the subtree. Therefore, the first criterion for the choice of the subtree should be the minimal enlargement in overlap, regardless of the type of the subnodes. Consequently, the XO-tree does not distinguish between data and directory nodes during this decision. Instead, it uses the minimal enlargement in overlap and in case of a tie, the minimal enlargement in volume and the volume itself as criteria for the decision. This should result in less overlap in the index and therefore a better query performance. The entire algorithm for the choice of the appropriate subtree is shown in figure 33.

As the XO-tree does not use reinserts to avoid splits, an overflow of a node leads either to a split or the creation of a supernode. First, a topological split is determined. If the overlap of the resulting nodes exceeds a certain threshold (usually 40% of the combined volume), an overlap-minimal split is determined. If this, in turn, yields an unbalanced split, the current node is extended by one logical disk block and therefore becomes a supernode. Supernodes are also split if this is possible within the above mentioned borders. For supernodes, however, the likelihood for a successful split is smaller than for ordinary nodes, as they were created because of their already high overlap between their entries. Nevertheless, they have to be split, if possible, to achieve an index structure that is as hierarchical as possible. Of course, the split of a supernode usually results in the creation of another supernode.

The topological split that is determined in the first stage of the split algorithm, is taken over from the X-tree and R*-tree respectively, but any split algorithm that minimizes the overlap between the two resulting nodes could be used. The algorithm can be divided into two phases, where in the first phase, the split axis is chosen and in the second one, the split plane is determined. To find the global optimum for the distribution of the elements into two nodes, would require an algorithm with exponential time complexity. This is unacceptable for any real world application. Therefore, an approximation of the optimal result is determined, using the following method. First, the entries of the node are sorted according to the lowest and according to the highest value of their approximation along each axis. Afterwards, for each of these sorting, *(M-2m+2)* distributions, where *M* denotes the maximal number of entries in the current node and *m* denotes the minimal number of entries in a node, are created. Obviously, *M* and *m* vary depending on the number of blocks a node occupies. We choose *m* to be 40% of *M*. The *k*-th distribution $(k \in \{1,\ldots,(M-2m+2)\})$ is described as follows: The first group contains the first

(m-1)+k entries, whereas the second group contains the remaining elements of the sorting. For each of these distributions, the sum of the margins of the approximations is calculated. The axis with the smallest sum of all the margin values of its distributions is chosen as split axis. To determine the best distribution along the split axis, the entries are again sorted along the split axis according to their lowest and highest value. Again the same number of distributions is determined and the one, which yields the smallest overlap between the two groups, is chosen. Afterwards, the node is split according to this distribution. The algorithms are shown in figure 15 and 16 (chapter 2).

An overlap-minimal split is determined, if the overlap of the topological split exceeds a threshold. The algorithm was taken over from the X-tree. It has the topological split algorithm as a basis. First, the potential split axes are determined. They are the axes according to which all subnodes of the current node have already been split and can be identified by examining the split-histories of those subnodes. Afterwards a topological split is performed, which only considers the potential overlap-minimal split axes. Of course, the minimal number of entries in a node *m* is now one. Otherwise, the overlap-minimal split would not test any distribution that has not been already considered by the topological split algorithm. This way the overlap-minimal split is found, which may be an unbalanced split. A split is considered unbalanced, if one node is less than 40% filled.

If an overlap-minimal split results in an unbalanced situation, the node is not split, but extended by one disk block. This way, a normal node gets a supernode. Of course, the disk blocks a supernode occupies, should be successive ones on the secondary storage. Otherwise, the intended effect that the node can be linearly scanned, is not achieved. Therefore, an implementation of the XO-tree has to ensure this either by pre-allocating empty disk blocks or by moving the node onto an area of sufficient size.

```
PageAdr OverlapMinimalSplit(PageAdr pa)
{
    uint i, k; // dimension and distribution indexes
    bitvector      split_vector; // contains the intersection of all split histories
    uint num_axis = 0; // number of potential split axes
    uint axis[dimension]; // potential split axes
    Page p = LoadPage(pa);
    Distribution CurrentDistribution, BestDistribution;
    PageAdr new_brother;
    split_vector = p.childnode[0].splithistory;
    for (i = 1; i < p.num_objects; i++)
        split_vector = split_vector & p.childnode[i].splithistory;

    for (i = 0; i < dimension; i++)
    {
        if ((split_vector[i] = TRUE)
            axis[num_axis] = i;
            num_axis = num_axis + 1;
    }
    for (k = 0; k < num_axis; k++)
    {
        CurrentDistribution = TopologicalSplitAccordingAxis(axis[k], pa);
        if (Overlap(CurrentDistribution) < MinOverlap)
            MinOverlap = Overlap(CurrentDistribution);
            BestDistribution = CurrentDistribution;
    }
    new_brother = SplitAccordingTo(BestDistribution);
    return new_bother;
}
```

Figure 34: Overlap-minimal split algorithm.

4.4.2 Delete

The delete algorithm of the XO-tree is very straightforward and simple. First, a point query with the object that has to be deleted is performed to identify the data node that contains this object. Afterwards the object is deleted from this node and the approximation of the node is adapted. If the deletion of the object causes an underflow for the data node, all entries from that node are distributed among its brothers. This may in turn cause overflows of the brother nodes, which are treated as in the insertion algorithm. The deletion of a node may lead to the underflow of its father node, if the father node is an ordinary directors node. If the father is a

supernode, it may be shortened by one block and even become an ordinary directory node. If the last but one element of the root is deleted, the root can be entirely deleted and the remaining entry is included in the union of the nodes on the former second level. If this happens, the tree shrinks by one level. Figure 35 shows a simplified version of the algorithm, where the part that operates on data nodes is omitted.

```
bool Delete(PageAdr pa, Point object)
{
    int i;
    bool underflow;
    Page p = LoadPage(pa);
    /* choose the subtree from which the object has to be removed */
     for (i = 0; i < p.num_objects; i++)
    {
        if (IsPointInRegion(object, p.region[i]))
            underflow = Delete(p.childpage[i], object);
    }
    if (underflow)
    {
        /* if son underflowed: delete son node and redistribute its entries */
        p.RemoveEntry(i);
        p.num_objects = p.num_objects – 1;
        DistributeEntries(i);
        /* check for underflow in current node */
        if (p.num_objects < MIN_FANOUT)
            return TRUE;
    }
    return FALSE;
}
```

Figure 35: Delete-Algorithm of the XO-tree.

4.4.3 Update

Analogously to the delete algorithm, the update algorithm performs a point query for the object to determine the data node that contains it. If the updated object is still inside the page region of its node, the values are changed and the algorithm terminates. Otherwise, the object is deleted from the node and it is tested if the object is inside the page region of the father node. If so, it is inserted into the appropriate node, using the insertion algorithm. This process may continue along the insertion path up to the root. The algorithm is depicted in figure 36. Again, a simplified version is shown, which omits the part that operates on data pages.

```
bool Update(PageAdr pa, Point object)
{
      int i;
      bool ok;
      Page p = LoadPage(pa);
      /* choose the subtree to update the object */
       for (i = 0; i < p.num_objects; i++)
      {
            if (IsPointInRegion(object, p.region[i]))
                  ok = Update(p.childpage[i], object);
      }
      if (!ok)
      {
            /* check if object is inside of this nodes page region */
            if (IsPointInRegion(object, p.PageRegion))
                  Insert(object, pa);
            else
                  return FALSE;
      }
      return TRUE;
}
```

Figure 36: Update-Algorithm of the XO-tree.

5 Experimental Results

To verify the practical relevance of the XO-tree, it was implemented and tested with artificial and real data. These tests also provided information about the properties of the three different forms for page regions the XO-tree can use.

5.1 Experimental Setup

For the tests, the XO-tree was implemented in C++ with the use of templates. This ensured that an easy change of the approximation type that is used, was possible. Besides, different types of data could be handled. The X-tree implementation was adapted to also use the HS-algorithm for nearest neighbor queries. This was necessary, as the original X-tree implementation uses the RKV-algorithm for this query type. Unfortunately, the available implementation of the SR-tree could not be adapted in this way. Therefore, a direct comparisons between the SR-tree and the XO-tree was impossible.

All tests were performed on HP-C160 with at least 256Mbytes of RAM. The implementation of the XO-tree as well as the implementation of the X-tree allow point queries, range queries, nearest neighbor and k-nearest neighbor queries. The following test data was used:

- Artificial uniformly distributed point data of various dimensions (2, 3, 5, 10, 15, 20, 25, 30). Each of these data sets contained 10,000 points.
- 20-dimensional real point data from a astronomical database. This data set contained 100,000 points.

For all tests, the block size of the index structures was fixed at 4096 bytes. There was no caching mechanism, which would influence the performance measurements, implemented.

The performance of both index structures was tested with all the query types that were supported. For each query type the artificial as well as the real data was used for the tests. Each value presented is the average value of 1000 queries with different query points but otherwise identical parameters.

5.2 Results

5.2.1 Point Query

Figure 37 and figure 38 show the number of pages accessed during the processing of point queries for the uniformly distributed data. Figure 37 shows the results for successful queries, whereas figure 38 depicts the results for unsuccessful queries.

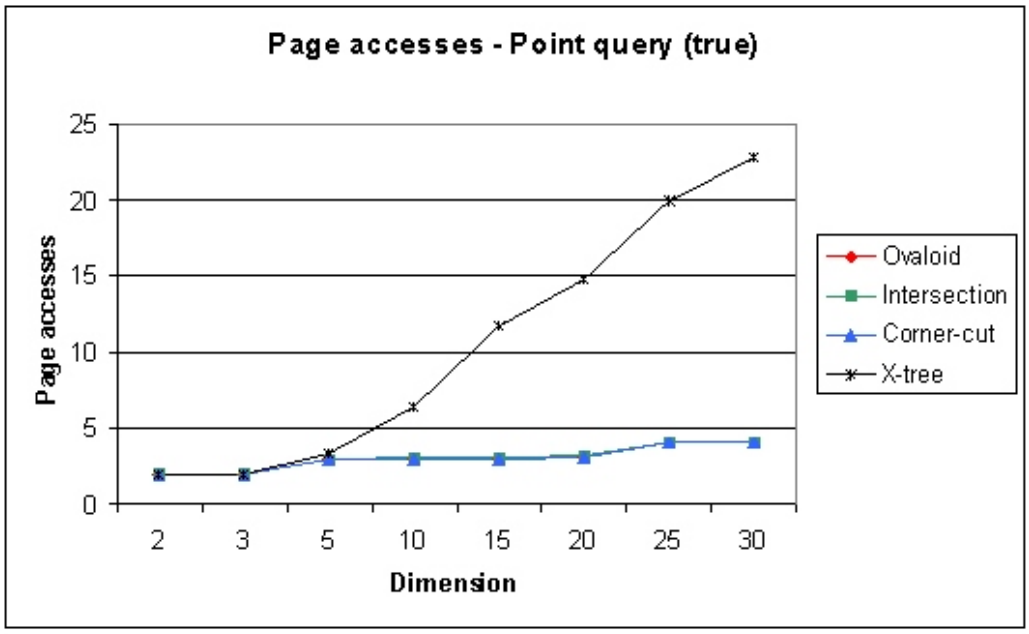


Figure 37: Point Queries – successful (Artificial Data).

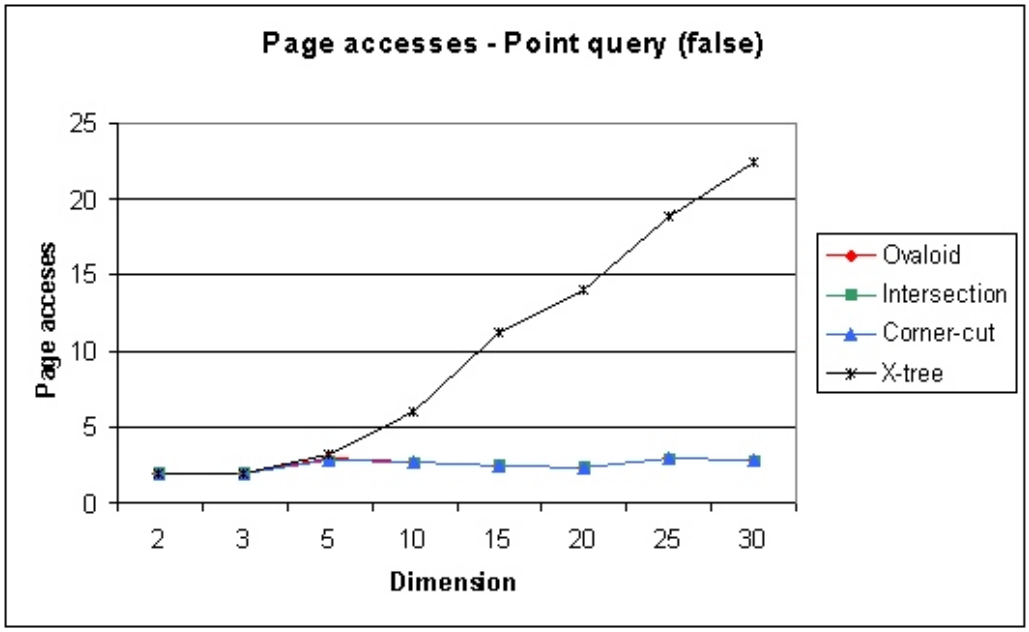


Figure 38: Point Queries – unsuccessful (Artificial Data).

For up to five dimensions, the performance of all XO-tree variants and of the X-tree is almost identical. However, for higher dimensions, all three XO-tree variants need fewer page accesses than the X-tree. The performance gain of the XO-tree variants gets even larger with growing dimension. The reason for this is the high overlap in the X-tree directory for high-dimensional data. This explanation is supported by figure 39, which shows the number of directory page accesses for point queries. Obviously, the overlap in the X-tree directory leads to many directory page accesses.

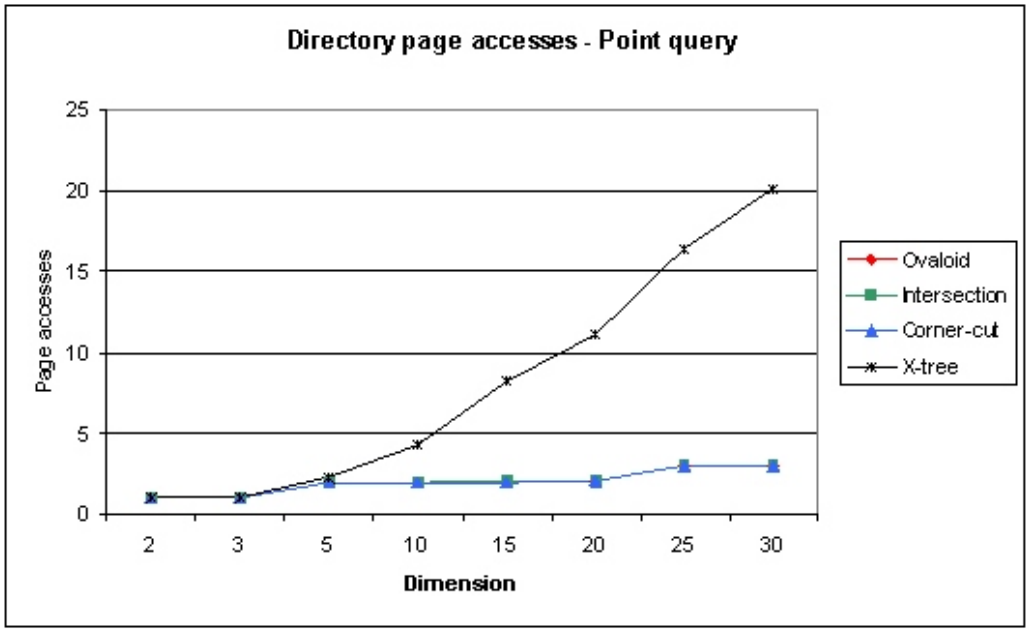


Figure 39: Point Query – Directory Page Accesses (Artificial Data).

On the other hand, the number of page accesses is almost identical with the height of the index for the XO-tree. As a result, the XO-tree in all its variants is up to 5.7 times faster than the X-tree for point queries, as can be seen in figure 40.

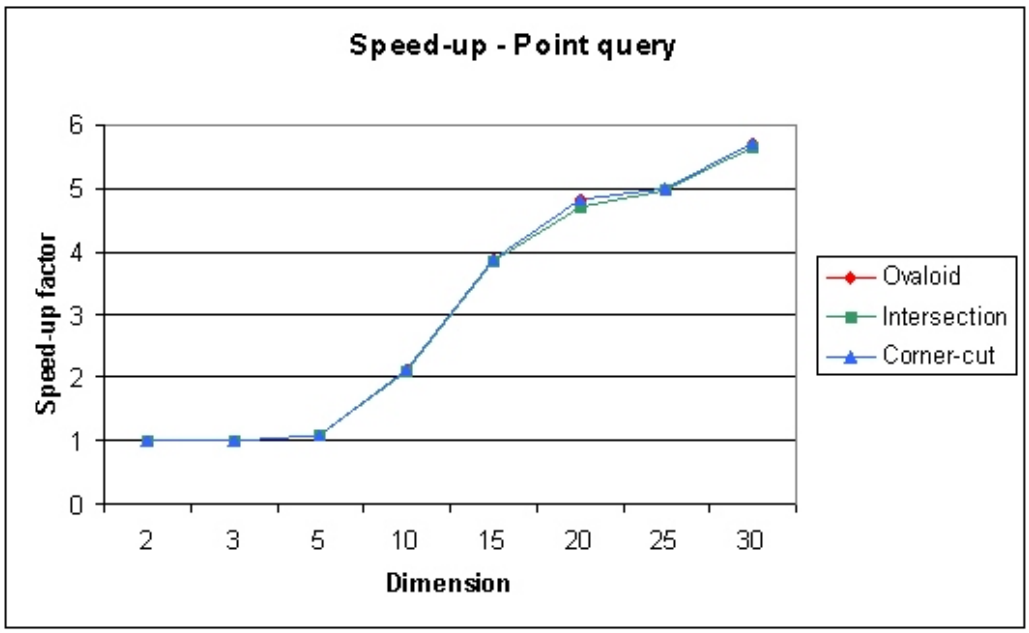


Figure 40: Point Query – Speed-Up (Artificial Data).

For the real data the speed-up factors that are shown in figure 41 are still more than two.

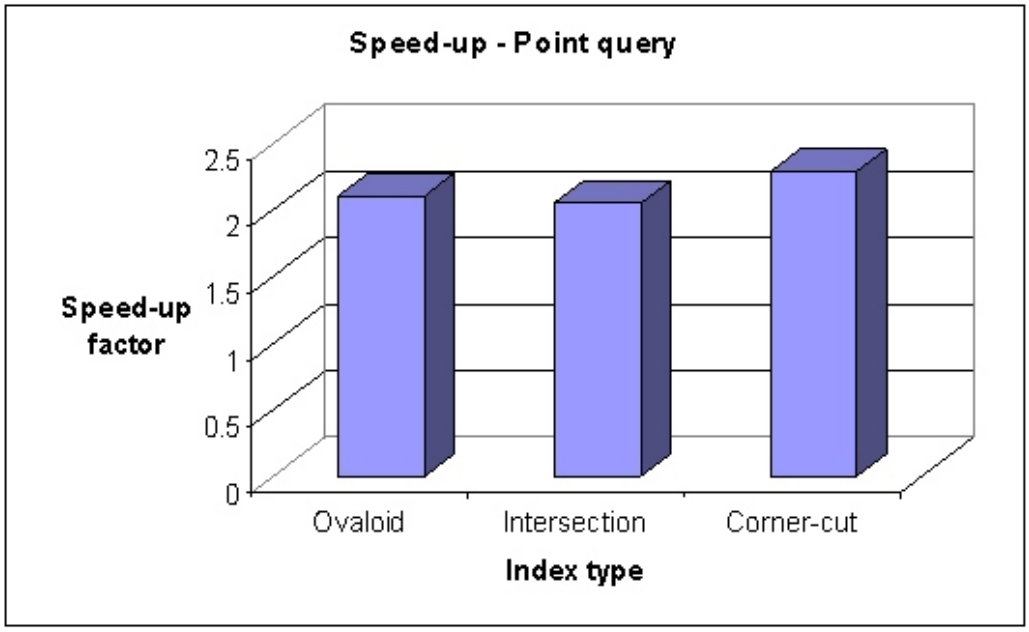


Figure 41: Point Query – Speed-Up (Real Data).

5.2.2 Range Query

In a second series of experiments, the performance of the index structures were measured using range queries. In figure 42, the number of page accesses for the artificial data and various dimensions is shown. Obviously, the XO-tree with the intersection of the ovaloid and the MBR as page region shows the worst performance in this situation. On the other hand, the XO-trees with ovaloids and corner-cut approximation as page regions outperform the X-tree. Again this performance gain gets larger with the number of dimensions of the data space.

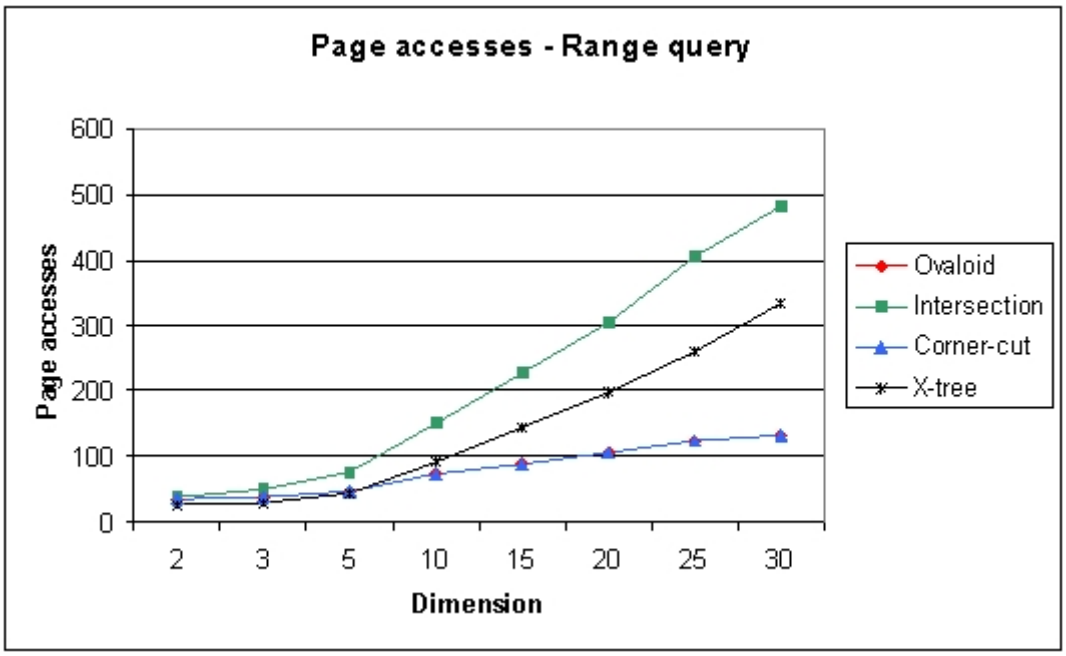


Figure 42: Range Query (Artificial Data).

The speed-up factors for this query type are depicted in figure 43. A factor of 2.5 is reached for 30-dimensional uniformly distributed data points.

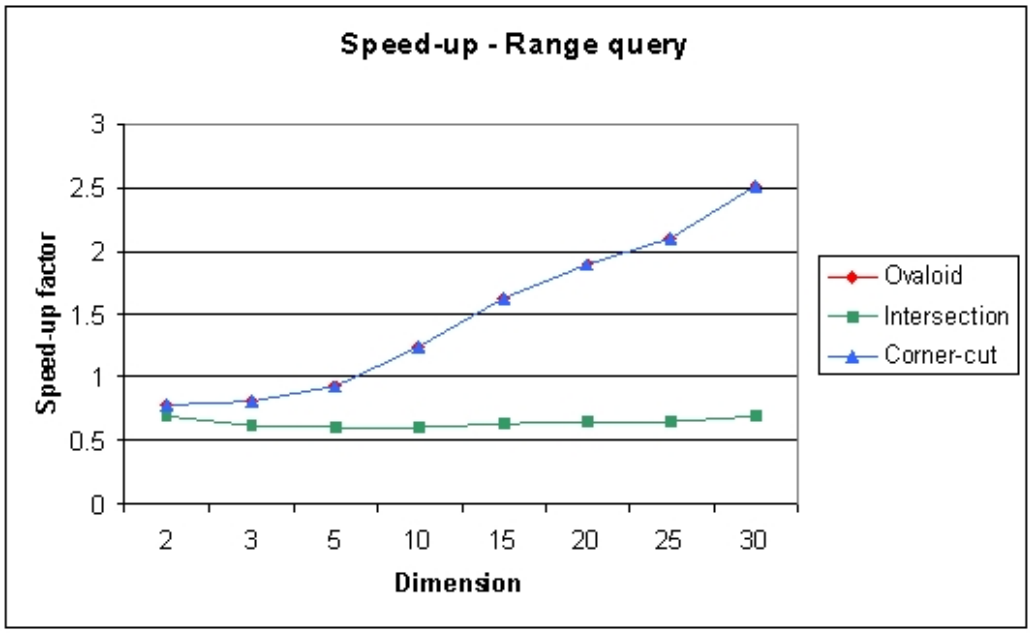


Figure 43: Range Query – Speed-Up (Artificial Data).

For the real data set, the number of page accesses for different query radiuses are shown in figure 44. Again the XO-tree with the intersection of an ovaloid and a

MBR as page region shows the worst performance. The other two XO-tree variants again outperform the X-tree. With growing query radius, the difference between the XO-tree and X-tree diminishes. This is not surprising, as the answer set contains more points for larger radiuses. As at least all data pages containing the points in the answer set and the directory pages on the paths to those nodes have to be read, the possibility for performance optimizations are reduced in the same amount as the size of the answer set increases.

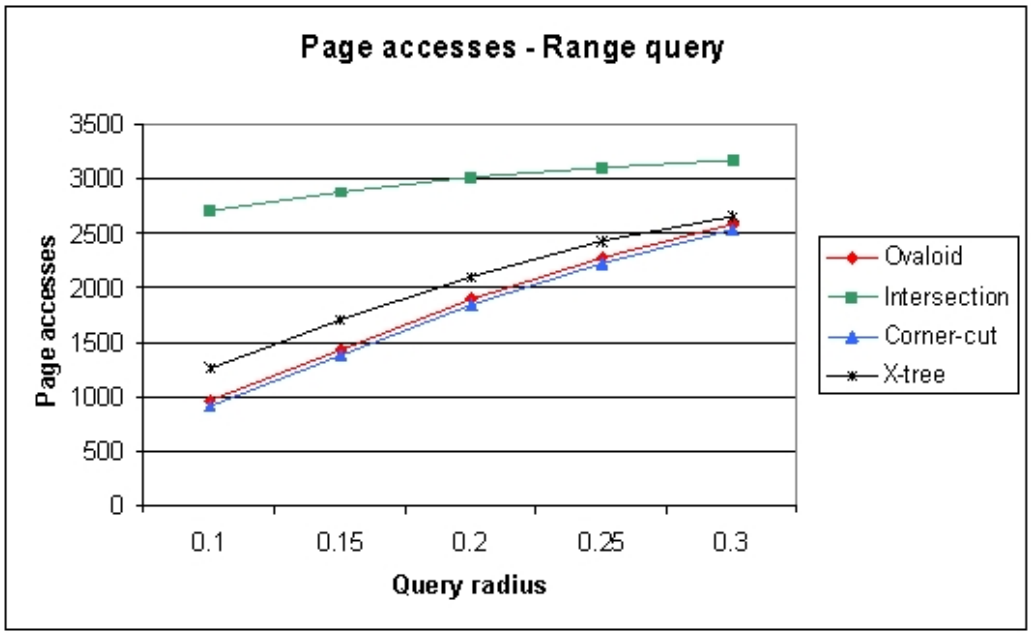


Figure 44: Range Query (Real Data).

5.2.3 Nearest Neighbor Query

As mentioned earlier, nearest neighbor queries are important in the context of similarity search systems. Therefore, the performance of the XO-tree and the X-tree was compared for this query type, too. The tests with the uniformly distributed point sets showed interesting results. If the query points were also stored in the database, the XO-tree variant, which uses the intersection between an ovaloid and a MBR as page region, needs considerably more page accesses than all the other structures. Nevertheless, the remaining two XO-tree variants outperform the X-tree. In fact, the number of page accesses that are needed to find the nearest neighbor to a query point, is similar to the number needed for a point query. This is true for the X-tree,

too. Consequently, the XO-tree is up to 3.5 times faster than the X-tree in these situations. Figure 45 shows this results.

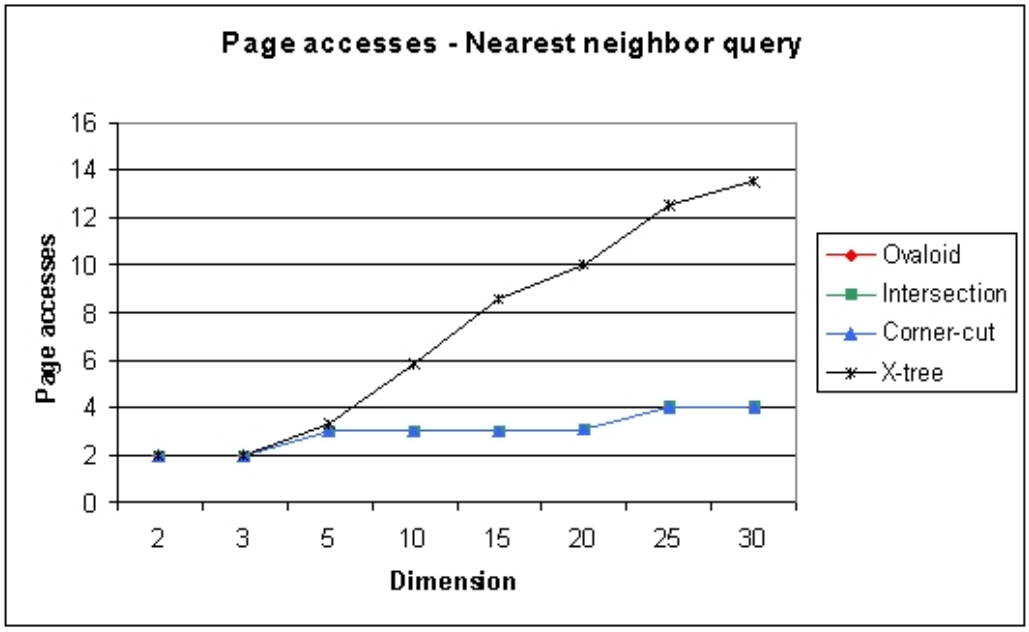


Figure 45: Nearest Neighbor Query 1 (Artificial Data).

If the query points were not stored in the database, different results are obtained. In this case, all the tested index structures show a similar performance, as depicted in figure 46. In data spaces with dimensions up to 20, the XO-tree variants with the corner-cut approximation or ovaloids as page regions are up to 30% faster than the X-tree. In even higher dimensional data spaces, the X-tree need fewer page accesses. This is again due to the fact, that the entire index has to be read, which gives the X-tree with its smaller directory an advantage.

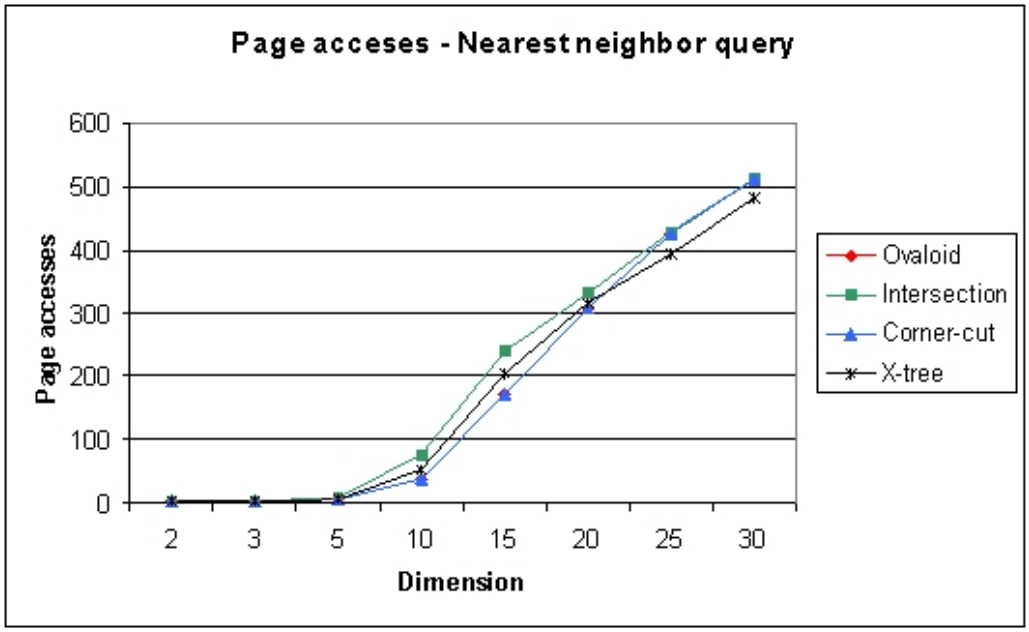


Figure 46: Nearest Neighbor Query 2 (Artificial Data).

For the real data, the fact that the query point was stored in the database, had no effect on the performance. As figure 47 shows, the XO-tree variant that uses the intersection of an ovaloid and a MBR as page region needed again more page accesses than the other XO-tree variants.

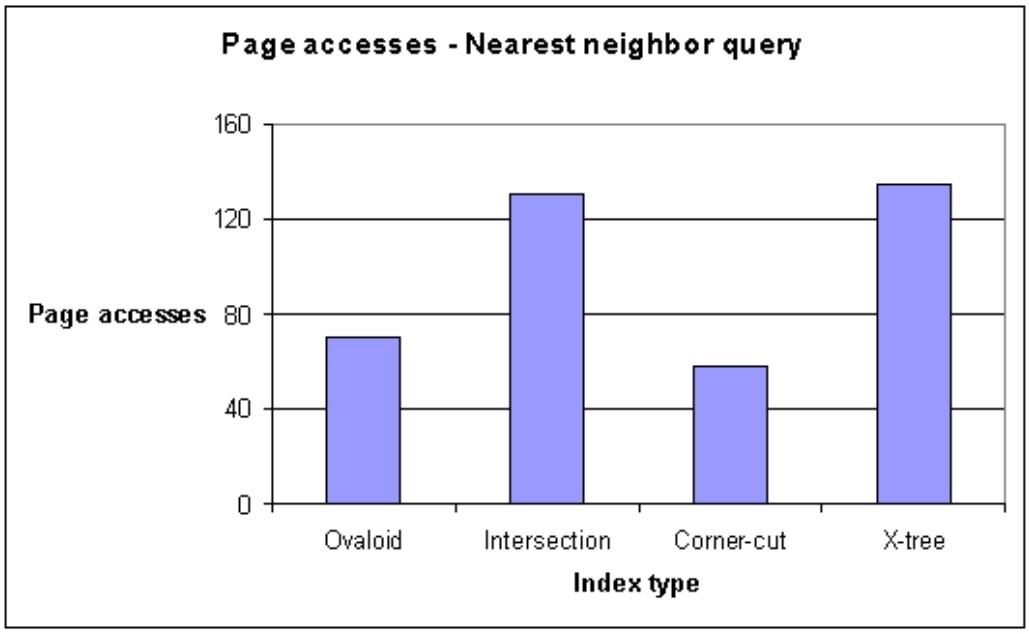


Figure 47: Nearest Neighbor Query 1 (Real Data).

Nevertheless, the other two XO-tree variants outperform the X-tree by a factor of two, as depicted in figure 48.

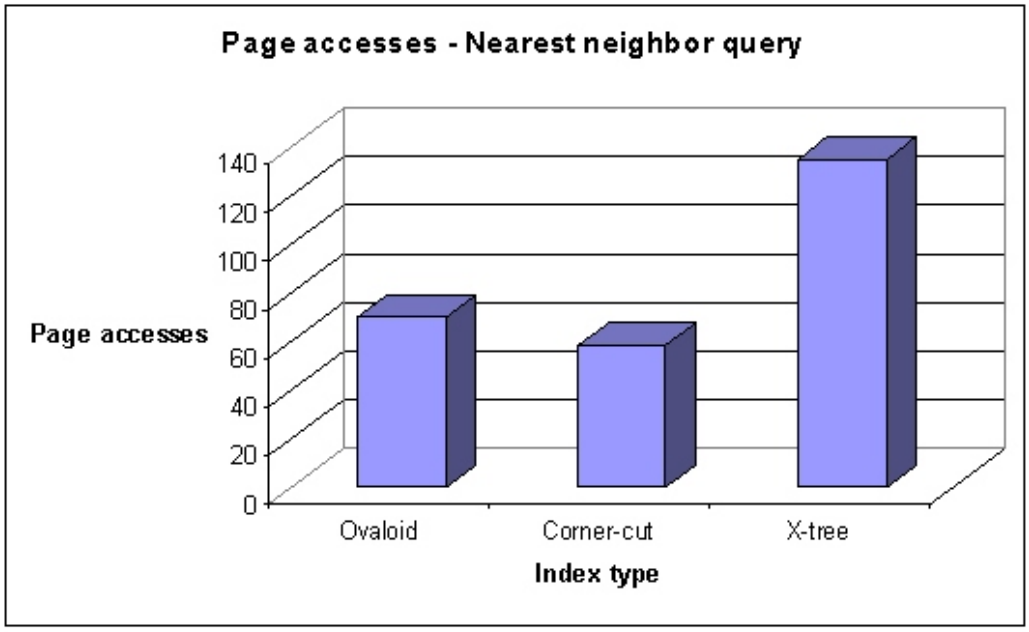


Figure 48: Nearest Neighbor Query 2 (Real Data).

5.2.4 K-nearest Neighbor Query

Finally, tests with k-nearest neighbor queries were made. Figure 49 shows the number of page accesses that were necessary to process a k-nearest neighbor query, which returns an answer set that has a size of 0.05% of the database size. Obviously, the results are similar to those for the nearest neighbor queries with query points that were not stored in the database. Again, the X-tree had advantages in data spaces with more than 20 dimensions due to its smaller index. In data spaces with lower dimensionality, all XO-tree variants needed less page accesses than the X-tree.

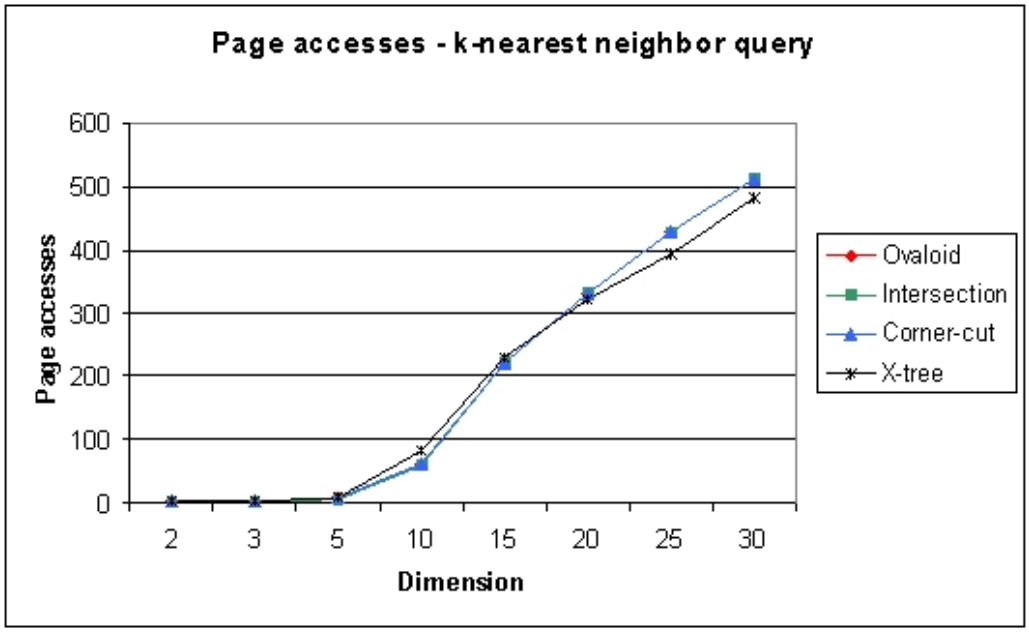


Figure 49: K-Nearest Neighbor Query (Artificial Data).

For the real data set, the results are presented in figure 50. The two XO-tree variants, which use ovaloids or the corner-cut approximation as page regions, need about 30% fewer page accesses than the X-tree. The intersection variant of the XO-tree, however, shows almost the same performance as the X-tree. Of course, the number of page accesses increases with the size of the answer set.

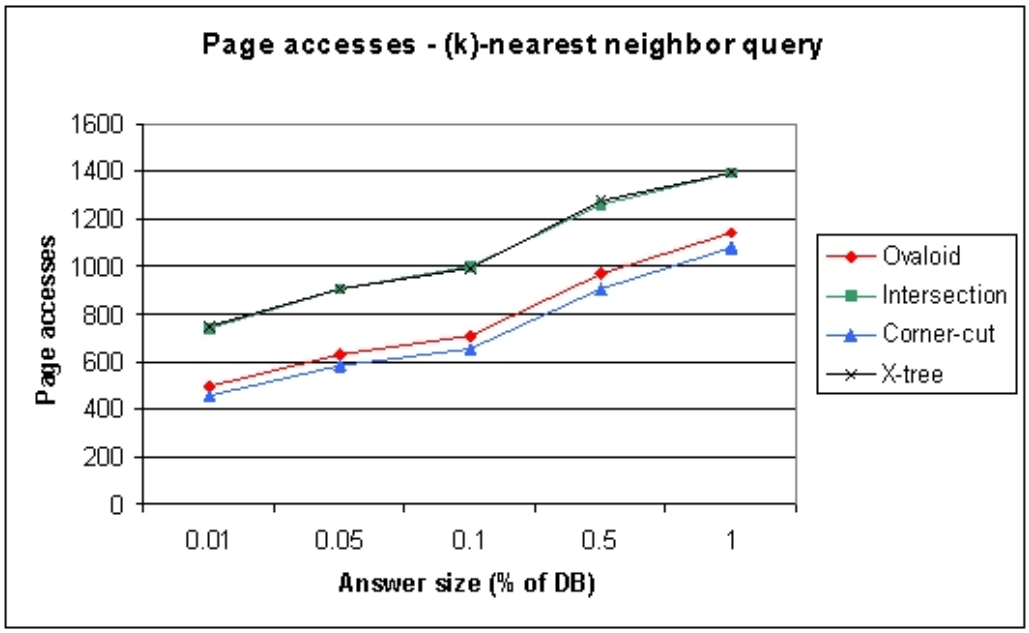


Figure 50: K-Nearest Neighbor Query (Real Data).

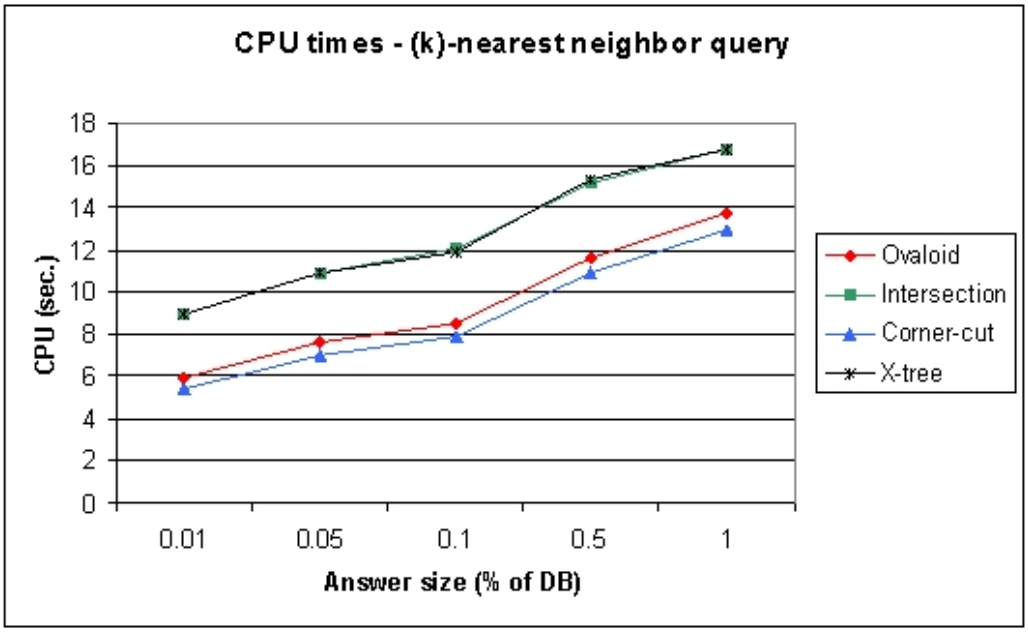


Figure 51: CPU Times (Real Data).

Finally, figure 51 shows the CPU times needed to process the k-nearest neighbor queries for the real data. A comparison with the previous figure reveals that the higher complexity for the basic operations in the XO-tree does not affect the query processing time significantly.

5.3 Summary

The performance of the XO-tree with all its variants was tested thoroughly and compared with the results of the X-tree. It turned out that the XO-tree variant, which uses the intersection between an ovaloid and a MBR as page region, shows bad performance for (k)-nearest neighbor queries and range queries. This supports the considerations about this form of page regions as presented in section 4.2.2. The other two XO-tree variants, on the other hand, usually outperform the X-tree for all query types. Only in cases where the entire directory is read, the X-tree performs better than the XO-tree. In these cases, however, a hierarchical index is always inferior to the linear scan due to the overhead of accessing the directory pages. Therefore, the use of an index is unjustified in these cases anyway.

Of the two XO-tree variants that outperform the X-tree, the one that uses the corner-cut approximation as page region, usually showed a moderately better

performance. Surprisingly, the error in the distance calculation of this approximation type did not affect its performance negatively.

6 Conclusion and Future Work

6.1 Conclusion

As we saw in chapter one, many applications like CAD, multimedia, molecular biology and time sequence analysis require high-dimensional indexing techniques. For the similarity search, the objects are transformed into points in a high-dimensional vector space using a feature transformation. Those vectors can only be managed efficiently with the use of multidimensional index structures.

In chapter two, we introduced and discussed several of such index structures. This were the R^*-tree, the X-tree, the SS-tree, the SR-tree and the TV-tree. Their important properties were described and the advantages and disadvantages of each structure were examined. The type of page region that is used in an index structure as well as the optimization of the overlap in the index were identified as important factors for the performance of an index structure in high-dimensional data spaces.

Different forms of page regions and their properties were discussed in the third chapter. The page region in an index structure should minimize the dead space, i.e. the area covered by the page region but not by any object inside the page region, in order to minimize the overlap in the directory. During this optimization process, two major properties of the page region have to be watched closely. First, the size of the description of the page region should be minimized. Otherwise, the height of the directory deteriorates and the performance of the index structure suffers. The second important point is the complexity of the basic operations for the page region. Those basic operations are used by the query processing algorithms and must therefore be executed efficiently to yield a good query performance.

Based on the results of the previous chapters, the XO-tree was developed in chapter four. As basis for this new index structure, the X-tree was chosen. Three new forms of page regions were introduced with the ovaloid, the intersection of an ovaloid and a minimal bounding rectangle and the corner-cut approximation. All of those page regions were designed to minimize the dead space, while keeping an eye on the complexity of the description and of the basic operations. The insertion algorithm was modified to optimally support the aim of minimizing the overlap in the directory.

Experiments with artificial and real data showed that the XO-tree outperforms the X-tree for almost every query type. The speed-up over the X-tree usually yields a factor of about two. The experiments also showed that the intersection of an ovaloid and an axis-parallel minimal bounding box does not yield the desired performance for similarity queries. The ovaloid and the corner-cut approximation, on the other hand, yield equally good performance for every query type. The use of these forms of page regions allows efficient indexing of high-dimensional data spaces, especially in the range between ten and twenty dimensions.

6.2 Future Work

Due to problems with the available implementation of the SR-tree, a comparison between the SR-tree and the XO-tree could not be made. However, as the SR-tree represents a similar approach as the XO-tree, such a comparison would be interesting. Therefore, this should be done after solving the problems with the different implementations. A comparison with a bulk-loading variant of the X-tree might also be of interest.

The topological split algorithm of the XO-tree is inherited from the R^*-tree. Here may be some room for optimization left, if special properties of high-dimensional vector spaces are considered.

Finally, the practical applicability of the XO-tree must be examined closely. Current commercial database systems are extended from the pure relational system towards so-called object-relational systems. This opens up the possibility to integrate new index structures, like the XO-tree, into those systems.

Appendix

A List of Figures

B List of Definitions

C Reference

[Abl 93] Ablaßmeier K.: *'Erweiterung eines geometrischen Anfrageprozessors um approximationsbasierte Anfragen und Operationen'*, Master Thesis (in German), Technische Universität München, 1993.

[AFS 93] Agrawal R., Faloutsos C., Swami A.: *'Effecient Similarity Search In Sequence Databases'*, Proc. 4th Int. Conf. on Foundations of Data Organization and Algorithms, 1993, LNCS 730, pp. 69-84.

[ALSS 95] Agrawal R., Lin K., Shawney H., Shim K.: *'Fast Similarity Search in the Presence of Noise, Scaling , and Translation in Time-Series Databases'*, Proc. of the 21st Conf. on Very Large Databases (VLDB), 1995, pp. 490-501.

[Ben 75] Bentley J. L.: *'Multidimensional Search Trees Used for Associative Searching'*, Communications of the ACM, Vol. 18, No. 9, 1975, pp. 509-517.

[Ben 79] Bentley J. L.: *'Multidimensional Binary Search in Database Applications'*, IEEE Trans. Software Eng. 4(5), 1979, pp. 397-409.

[Ber 97] Berchtold S.: *'Geometry based search of similar parts'*, (in german), Ph.D. thesis, University of Munich, 1997.

[BKK 96] Berchtold S., Keim D., Kriegel H.-P.: *'The X-tree: an Index Structure for High-Dimensional Data'*, Proc. of the 22nd VLDB Conf., Bombay, India, 1996, pp.28-39.

[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 Managment of Data, Atlantic City, NY, 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.

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

[BO 97] Bozkaya T., Ozsoyoglu M.: *'Distance-Based Indexing for High-Dimensional Metric Spaces'*, Proc. 1997 ACM SIGMOD In. Conf. on Management of Data, Tucson, AZ, 1997.

[Bri 95] Brin S.: *'Near Neighbor Search in Large Metric Spaces'*, Proc. 21st VLDB Conference, 1995, pp. 574-584.

[Chi 94] Chiueh T.: *'Content-Based Image Indexing'*, Proc. 20th VLDB Conference, 1994, pp. 582-593.

[Com 79] Comer D.: *'The Ubiquitous B-tree'*, ACM Computing Surveys 11(2), 1979, pp.121-138.

[CPZ 97] Ciaccia P., Patella M., Zezula P.: *'M-tree: An Efficient Access Method for Similarity Search in Metric Spaces'*, Proc. 23rd Int. Conf. on Very Large Databases (VLDB ´97), Athens, Greece, 1997.

[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.

[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, San Francisco, CA, 1987.

[GM 93] Gary J. E., Mehrotra R.: *'Similar Shape Retrieval using a Structural Feature Index'*, Information Systems, Vol. 18, No. 7, 1993, pp. 525-537.

[Gut 84] Guttman A.: *'R-trees: a dynamic index structure for spatial searching'*, Proc. ACM SIGMOD Int. Conf. on Managment of Data, Boston, MA, 1984, pp. 47-57.

[Hin 85] Hinrichs K.: *'Implementation of the Grid File: Design Concepts and Experience'*, BIT 25, pp. 569-592.

[HS 95] Hjaltson G. R., Samet H.: *'Ranking in Spatial Databases'*, Proc. of the 4th Symposium on Spatial Databases, Portland, ME, 1995, pp.83-95.

[HSW 88a] Hutflesz A., Six H.-W., Widmayer P.: *'Globally Order Preserving Multidimensional Linear Hashing'*, Proc. 4th IEEE Int. Conf. on Data Engineering, 1988, pp. 572-579.

[HSW 88b] Hutflesz A., Six H.-W., Widmayer P.: *'Twin Grid Files: Space Optimizing Access Schemes'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1988.

[Jag 91] Jagadish H. V.: *'A Retrieval Technique for Similar Shapes'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, pp. 208-217.

[Kei 97] Keim D. A.: *'Efficient Similarity Search in Spatial Database Systems'*, Habilitation thesis, Institute for Computer Science, University of Munich, 1997.

[KKS 98] Kastenmüller G., Kriegel H.-P., Seidl T.: *'Similarity Search in 3D Protein Databases'*, Proc. German Conference on Bioinformatics (GCB'98), Köln (Cologne), 1998.

[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 Managment of Data, Tucson, AZ, 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, 1995, pp. 517-542.

[MG 93] Mehrotra R., Gary J. E.: *'Feature-Based Retrieval of Similar Shapes'*, Proc. 9th Int. Conf. on Data Engeneering, 1993.

[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.

[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.

[RKV 95] Roussopoulos N., Kelley S., Vincent F.: *'Nearest Neighbor Queries'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, San Jose, CA, 1995, pp.71-79.

[Sei 97] Seidl T.: *'Adaptable Similarity Search in 3-D Spatial Database Systems'*, Ph.D thesis, University of Munich, 1997.

[SK 97] Seidl T., Kriegel H.-P.: *'Efficient User-Adaptable Similarity Search in Large Multimedia Databases'*, Proc. 23rd Int. Conf. on Very Large Databases (VLDB'97), Athens, Greece, 1997, pp. 506-601.

[TC 91] Taubin G., Cooper D. B.: *'Recognition and Positioning of Rigid Objects Using Algebraic Moment Invariants'*, Geometric Methods in Computer Vision, Vol. 1570, SPIE, 1991, pp. 175-186.

[Uhl 91] Uhlmann J.K.: *'Satisfying General Proximity/Similarity Queries with Metric Trees'*, Information Processing Letters, Vol. 40, 1991, pp.175-179.

[WJ 96] White D.A., Jain R.: *'Similarity Indexing with the SS-tree'*, Proc. of the 12th Int. Conf. on Data Engineering, New Orleans, LO, 1996, pp.516-523.

[Yia 93] Yiannilos P.N.: *'Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces'*. ACM-SIAM Symp. on Discrete Algorithms, 1993, pp.311-321.