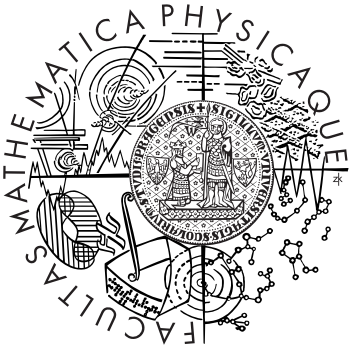
Charles University Prague,
Faculty of Mathematics and Physics

MASTER THESIS



Ondrej Danko

Elliptic Indexing of Multidimensional Databases

Department of Software Engineering

Advisor: Doc. RNDr. Tomáš Skopal, Ph.D.
Course of study: Software Systems

2008

At this point, I would like to thank my advisor for his valuable advice and all others who have helped me to reach the goal.

> Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism.
>
> —*Dave Barry*

I hereby proclaim that I worked out this thesis on my own, using only the resources stated. I agree that the thesis may be publicly available.

In Prague on March 20, 2008 Ondrej Danko

Contents

Abstract

Title	: Elliptic indexing of multidimensional databases
Author	: Ondrej Danko
Department	: Department of Software Engineering
Supervisor	: Doc. RNDr. Tomáš Skopal, Ph.D.
Supervisor's e-mail address	: Tomas.Skopal@mff.cuni.cz
Keywords	: multidimensional data, indexing, R-tree, minimum volume covering ellipsoid

Abstract: In this work variation of R-tree, which hierarchically partition indexed space using minimum volume covering ellipsoids (MVCE) instead of usually used minimum bounding rectangles, is presented.

Main aspects, which determine R-tree index structure performance, are studied from the available resources at the beginning. Base on this studies "eR-tree" (**e**llipsoid **R-tree**) is designed. Afterward algorithms of MVCE construction are carefully analyzed, as the choice of the algorithm is crucial for the efficiency of indexing and retrieval. At the end of the work, eR-tree implementation over ATOM framework is presented along with experiments done on synthetic and real data sets.

Název práce	: Eliptické indexování vícerozměrných dat
Autor	: Ondrej Danko
Katedra	: Katedra softwarového inženýrství
Vedoucí diplomové práce	: Doc. RNDr. Tomáš Skopal, Ph.D.
e-mail vedoucího	: Tomas.Skopal@mff.cuni.cz
Klíčová slova	: viacerozmerné data, indexácia, R-strom, minimálny ohraničujúcí ellipsoid

Abstrakt: V tejto práci je diskutovaná variácia R-stromu, ktorá k hierarchickému deleniu indexovaného priestoru využíva namiesto bežne používaných minimálne ohraničujúcich obdĺžnikov minimálne ohraničujúce ellipsoidy (MVCE).

V úvode práce sú zosumarizované hlavné faktory ovplyvňujúce výkon R-stromových štruktúr z dostupných štúdií, čo vedie k niekoľkým variantám "eR-stromu" (**e**llipsoid **R-tree**). Následne sú rozobrané známe algoritmy konštrukcie ellipsoidov. Výber vhodnej metódy konštrukcie MVCE je jedným z kľúčových faktorov ovplyvňujúcich efektivitu indexácie a následného vyhľadávania. V závere práce je prezentovaná experimentálna implementácia eR-stromu nad frameworkom ATOM. Výhodnost použitia MVCE v R-strome je prezentovaná experimentami jak nad syntetickými datami, tak nad reálnymi.

Chapter 1

Introduction

In recent years demands for efficient searching in large multimedia data sets have begun emerging much more often than anytime before. Mostly the newest application in the areas like medicine, geography or CAD were lacking because of the absence of techniques which would enable them to search efficiently in protein databases, geographical maps or CAD datasets.
Each of the applications employing multimedia database usually manipulate different data "types" (proteins, geographical locations, pictures) which always requires a custom solution for storing and quering of data. To eliminate the need for a proprietary solution, a simple, yet powerful idea is applied: *feature transformation*. The idea is that the input data is transformed into *feature vectors* – multidimensional points of vector space. Afterward a mutual technique is used to store and search feature vectors of either medicine, geography or CAD data.

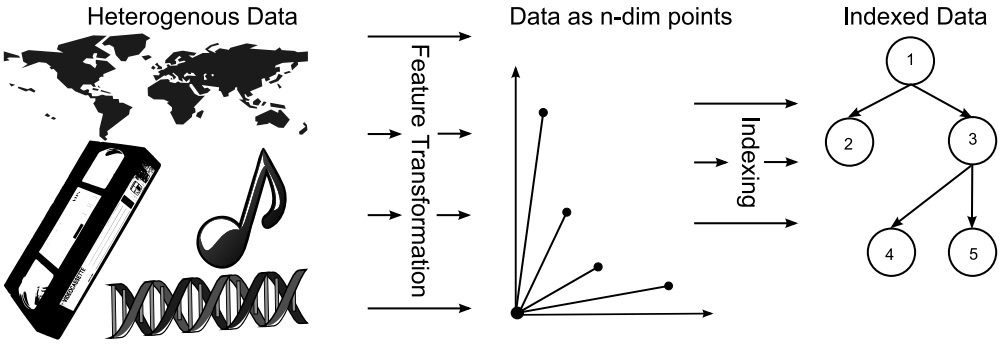


Figure 1.1: THe feature transformation.

To illustrate the use of feature transformation we can think of a task to index

collection of images. The very feature transformation can be defined as e.g. 16-color histogram of an image, which will define coordinates of the picture in 16 dimensional space. These feature vectors are afterward indexed. Now to search for images similar in color to a model image, we just query the index for the closest feature vectors to the feature vector of that model image. If we would like to search images for similar patterns, the only change required would be in the definition of the feature transformation.

Most common query types in multidimensional databases are *range queries* and *nearest-neighbor* (also known as *similarity*) queries.
Range query simply defines a portion of the vector space. Running it against the index it will return all feature vectors lying inside that portion.
Nearest-neighbor query accepts as an input a feature vector and optionally some metrics (e.g. euclidean, maximum or weighted). The result set returned will contain feature vectors, which lie closest to the input feature vector with respect to the given metrics.

Since early 80's there has been devoted a lot of energy to the development of an efficient method of multi dimensional data indexing. As the first successful approach can be considered Guttman's *R-tree* [Gut84] index structure. Later on, many different techniques were proposed and a lot of variations of the original R-tree were supplied for evaluation. Nowadays, largely the R* modification of the R-tree is used in leading database management systems to index multidimensional data.

1.1 Aim of the thesis

The aim of this work is to *investigate* the applicability of ellipsoids (4.1.2) to R-tree (2.2) index structure.

R-tree index structure and relevant algorithms should be accommodated to the use of ellipsoid regions instead of usual minimum bounding rectangles. Resulting structure should be a subject of extensive tests on synthetic and real data sets, proving pros and cons of ellipsoid regions in R-tree.

1.2 Outline of the work progress and paper organization

First, an extensive survey of the existing multi dimensional indexing techniques was done. The emphasis was put on many available R-tree variants. The outcome of the survey can by found in Chapter 2.

Afterward the time was devoted to the "ellipsoid" theory with focus on the algorithms of ellipsoid construction. Here was spent the biggest portion of the time devoted to the whole work, as it is crucial to choose an appropriate algorithm. An inappropriate algorithm could lead to a significant indexing speed degradation or poor retrieval results due to non-optimal ellipsoids (non-optimal in terms of the shape). To save the time, all the investigation work was carried out in matlab. As soon as the choice of the algorithm was clear, it was re-implemented to c++. The results of this phase can be found in Chapter 4.

In the next phase, using the *knowledge from the introductory survey* and *having the idea of the "efficiency" of the ellipsoid construction* eR-tree (– ellipsoid R-tree) was designed. The individual steps leading to the final design with some explanatory comments should be found in Chapter 3.

The implementation of eR-tree came out of the existing R-tree implementation over ATOM framework. The details of implementation are discussed in Chapter 5.

To evaluate the new eR-tree, extensive tests were carried out on synthetic and real data sets. The results and the explanations of measurements are presented in a separate Chapter 6.

Last but not least in Appendix A can be found a "visual" presentation of the eR-tree. We had frequently make use of visual examination. Appendix B can be of a great use when browsing through the attached compact disc.

Chapter 2

Multidimensional indexing

In this chapter, we will explain how can be standard relational indexing techniques used for index multidimensional data and what are their drawbacks. We will introduce a well-known term *curse of dimensionality*. Afterward we will list most important multidimensional index structures. Later our focus will be devoted to the R-tree index structure and its variants.

2.1 Relational indexes

In relational databases we can store multidimensional data[1] in a single table where each record will correspond to one point. The table would be comprised of an identifier column and one column per dimension to represent the position. Indexing such a table with single key B-tree would require creating an index for each column. This technique is acceptable when the dimension is low e.g. 1, 2 or 3. In higher dimensions, we will experience the consequences of a *curse of dimensionality*. The course of dimensionality could roughly be expressed as that the amount of data required to keep the same level of data density is with increasing dimension increasing *exponentially*. To consider, the reverse holds also: the data density decreases exponentially with the increasing dimensionality. For example if we imagine uniformly distributed 10 points in a line segment of the length 1, the density of points is 0.1. If we want to keep this density in two dimensional space with domains $\langle 0, 1)$, we will need 100 points, and if we move to cube, 1000 points would be

[1] We interpret multidimensional data as points of a n-dimensional vector space. See the feature transformation in Chapter 1. In the ongoing text we will use terms *data*, *points* and *point-data* interchangeably, meaning the same.

required. Now, the question is, how it effects the retrieve performance of the B-tree. Assuming that we have 100 uniformly distributed two dimensional points of the form $[x, y] \quad x, y \in \{0.0, 0.1, 0.2, \ldots, 0.9\}$ and we issue the following query against the index: `SELECT * FROM db WHERE x > 0.45 AND x < 0.55 AND y > 0.45 AND y < 0.55`. The `SELECT` statement should return one point back, lying at the position $[0.5, 0.5]$. However, seeking into a single key B-tree index holding x coordinates will return 10 points with coordinates $[0.5, y] \quad y \in \{0.0, 0.1, \ldots, 0.9\}$. The analogous holds for index containing y coordinates. This implies we need to touch 20 points to process the query, which is 20% of all the data. When we move to a three-dimensional space and we will keep the data density, we would need to index 1000 points. Again, seeking into the index holding x coordinates will return 100 points of form $[0.5, y, z] \quad x, y \in \{0.0, 0.1, \ldots, 0.9\}$. Together we need to touch 300 points which corresponds to 30% of all points. When moving to even higher dimension, the use of indexes would become an obstacle and full scan would be by much faster. See illustration in Figure 2.1.

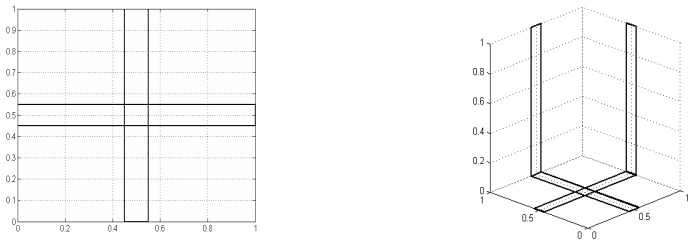


Figure 2.1: Touched points.

It is not true that the curse of dimensionality applies only to the case of multidimensional data indexing with relational techniques, in fact, spatial indexes are affected to. To cope with this problem spatial indexes are usually developed in favor of either low dimensions or high dimensions. As a low dimensional techniques we classify the original R-tree, R^+-tree, R^*-tree. For high dimensional data are the most suited TV-tree, X-tree, Pyramid tree, UB-tree or SS-tree.

2.2 R-Tree bases

In this section we describe the original Guttman's R-tree [Gut84]. It is a height-balanced tree similar to the B-tree. The basic idea behind is to hierarchically partition the search space into nested regions. Space partitioning

is neither complete nor disjoint. Nested hierarchies are formed as paths from the root node to the leaf nodes, where the root node region encloses the regions of its child nodes. See Figure 2.2. Searching in the R-tree index structure means traversing those paths of the tree that intersect with the search region.

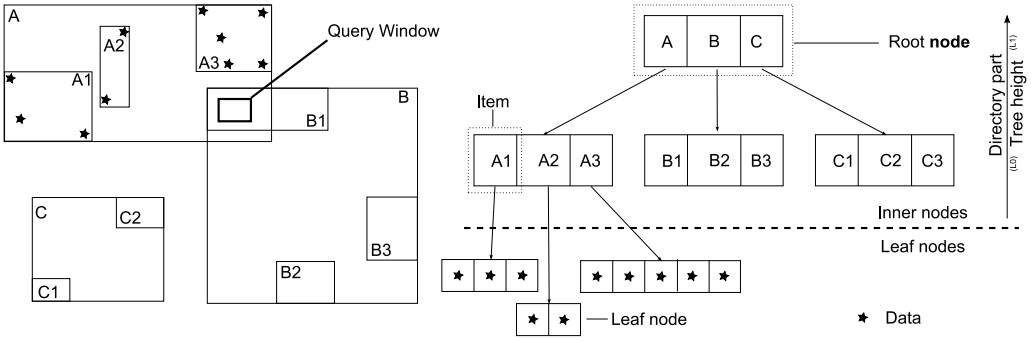


Figure 2.2: The R-tree structure. On the left there are 2 dimensional data bounded into nested regions, on the right there is presented the index structure.

The R-tree consists of two types of nodes: leaf nodes and inner nodes. The data is stored in the leaf nodes. A leaf node is comprised of simple *data pointers* which usually points to an external storage where data is actualy stored[2].
Inner nodes consist of a set of tuples[3] of the form *(I, child pointer)* where a child pointer is a pointer to another inner node or leaf node. In case child pointer points to another inner node, I represents the minimum bounding rectangle[4] of all items of the child node pointed by the child pointer. In case that the child pointer points to the leaf node, I represents MBR of all data stored in that leaf node.
The algorithm of insertion is following:

[2]We should notice, that original Guttman's work admits data of an arbitrary shape (not only points), therefore his leaf node contains except the data pointer also a minimum bounding rectangle of the data pointed by data pointer.

[3]We refer to a tuple of an inner node as to an item to be consistent with the framework terminology used to build an experimental implementation.

[4]We will use a minimum bounding rectangle, MBR, a minimum bounding box interchangebly.

Algorithm 1 R-tree insert

Require: P - point to be inserted

1: $LeafNode = Search(P)$ {Descend tree starting from the root node to the leaf node – leaf node which will accommodate new entry P. During the descend enlarge the regions of the relevant inner nodes if necessary. Return found LeafNode.}
2: **if** $LeafNode$ is ***not*** full **then**
3: insert P into $LeafNode$
4: **else**
5: $SplitLeafNode(LeafNode)$ {Create additional leaf node LLeafNode and split entries of LeafNode between original LeafNode and LLeafNode.}
6: insert P into either $LeafNode$ or $LLeafNode$
7: $PropagateSplitUpward()$ {Propagate leaf node split upward. Propagation can lead to inner node splits which are handled by SplitInnerNode(InnerNode) subroutine.}
8: **if** root node split occurred **then**
9: grow tree higher
10: **end if**
11: **end if**
12: **return**

In the step 1 we proceed in the way that the added new point will cause the least enlargement of MBRs. The choice of the leaf node, which will accommodate the new entry, and the splitting algorithm of overfulled leaf nodes are the crucial algorithms of the R-tree with a direct impact on the search performance. The performance of the R-tree with overlaying MBRs dramatically deteriorates though searching the tree with query window, which intersect overlaying regions, will result in multiple descends of the tree. See Figure 2.2, here we can notice that the rectangle A intersects with $B1$. If we issue range query (see *query window* in the figure) A, B and $B1$ need to be evaluated.
Guttman provides three types of splitting algorithms:

- Exhaustive split
- Quadratic-cost split
- Linear-cost split

*Exhaustive split*ting means to generate all the possible splits and choose the one that generates the minimum area. For node fanout, which mostly ranges

from 50-200 is this approach too expensive, though it would run in time $O(2^{fanout-1})$.
Quadratic-cost split in the first step chooses two points, which together form the MBR of the greatest area. These two points will be then placed to the different nodes(old one and new one). Afterward the algorithm proceeds with each other point and will assign it to the node by which the least enlargement will be caused. The algorithm is quadratic in number of points being split.
The *linear-cost* algorithm simply chooses two most distant points as a seeds and then assigns the rest of the points in the same way as the Quadratic-cost algorithm does.

For a comprehensive survey of R-tree indexing techniques [BBK01] is advised.

2.2.1 R*-tree

The extensive study of Guttman's R-tree on different data distributions led to the proposal of some optimizations as

- minimization of leaf level node region overlaps
- minimization of leaf level regions' surface
- minimization ofthe volume of inner nodes
- maximization of the storage utilization

R*-tree introduces *forced reinsert* since R-tree structures are highly susceptible to the order in which the element are inserted. When a leaf node becomes overfulled, some portion of points, which are the most distant from the centre of the node MBR, are deleted from index and afterward reinserted. This improvement brought the storage utilization up to 71% -76% . An R*-tree also prefers squared MBRs to rectangulars.

2.2.2 R+-tree

The key idea behind R+-tree is overlap free splitting in tree directory. Generally, there is no guarantee[5] that such a splitting exists.

In case there exists no overlap free splitting, the R+-tree introduces *forced splits*. Consider figure 2.2. To remove an overlap between the region A and

[5]When we are building index dynamically, it means, we don't know all points ahead.

the region B we necessarily need to split B1 region. Of course, in some situations forced splits need to be propagated until we reach leaf nodes, whereas the number of forced splits can exponentially increase till we reach them. Also as a side effect of forced splits R+-tree cannot guarantee 50% space utilization like the regular R-tree.

Authors of the R+-tree claim that their modification requires in average up to 50% less page access compared to the R-tree. For further details consider [SRF87].

2.2.3 SS-tree

An SS-tree was introduced to support similarity searches in higher dimensions (thus **S**imilarity **S**earch-**tree**). The SS-tree uses spheres instead of MBRs as page regions, see 2.3. When comparing properties of the MBRs to spheres it

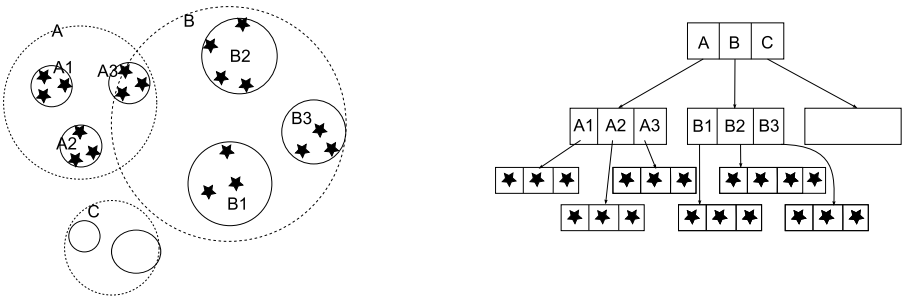


Figure 2.3: The SS-tree.

should be said that:

- Bounding spheres tend to produce regions bigger in *volume* than the MBRs.
- Bounding spheres tend to produce regions of smaller *diameter* than the MBRs.

Using spheres, the first property decreases the performance of range queries while the second is in favor of the nearest neighbor queries.

Another advantage of spheres is that they require less space to be stored than the MBRs. For an MBR we need to store two n-dimensional vectors, for a sphere it is enough to store one n-dimensional center and one-dimensional

diameter. This allows higher fanout of nodes, thus it eventually decreases the tree height.

Because of the performance reasons the spheres of SS-tree are not minimum volume spheres, but centroids. Thus the center of the sphere is computed as an average in each dimension of the points being bounded. The diameter is then calculated sin the way that it covers all the points. The SS-tree uses *forced reinserts* when the overflow condition is encountered. 30% of the points with the highest distance from the center of the sphere are reinserted (same as with the R*-tree). While the storage utilization of the R*-tree is just 70-75% , the SS-tree reaches in average 85%.

Splitting is based on variance. First, the dimension with the highest variance is chosen. Then the split plane orthogonal to that dimension is found so that the sum of variance in the new node and the old node is minimized.

Authors of the SS-tree claim that the insertion uses significantly less CPU time compared with the R*-tree (5-10x less). It is mainly because of the simplistic insertion algorithm and linear split algorithm compared with the quadratic split algorithm of the R*-tree.

The SS-tree outperforms the R*-tree approximately by the factor of two. More detailed performance comparison with the R*-tree can be found in [WJ96].

2.2.4 SR-tree

SR-tree is merely a combination of the SS-tree and the R*-tree. Authors provided in their work ([KS97]) an extensive comparison of MBRs and spheres properties. They define the region in SR-tree as an intersection of a MBR and a sphere. They believe, the region would be small in volume because of the MBRs and also small in diameter because of the spheres. This extension should bring a reasonable query performance for both range and nearest neighbor queries.

The insert and the split algorithm are taken from the SS-tree and are controlled solely by the spheres. The SR-tree slightly outperforms both the SS-tree and the R*-tree.

As the most significant inefficiency of this approach authors discuss storage requirements of the SR-tree region, which are one-and-a-half larger than of R*-tree and three times larger than of SS-tree.

Chapter 3

eR-tree

In this chapter, we will describe eR-tree. eR-tree is the name of our modification of Guttman's R-tree which name stands from ***e***llipsoid ***R-tree***.

3.1 Motivation

It was written in the section dealing with R-trees (2.2), the performance of the structure is mostly determined by the amount of the region overlaps and dead space coverage. We believe that ellipsoids help us to cope mainly with the second of the problems. Intuitively, an ellipsoid as a quadratic curve, will *cover* more tightly the data. To grasp our "motivation" consider Figure 3.1. Ten randomly generated points are covered with the ellipsoid and the MBR. The volume of the ellipsoid is 0.074937 and the volume of the MBR is 0.181156 which means that the ellipsoid is ≈ 2.4 times smaller in volume than the MBR.

3.2 eR-tree Regions

Here we describe three variants of the eR-tree regions, chronologically from the most straightforward variant to the Variant #3, which we have decided to implement and thoroughly test. We explain what the demerits of each of the variants are and how the consequent variant copes with these demerits.

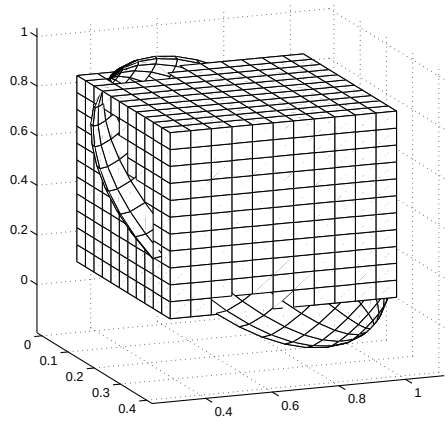


Figure 3.1: Covering points with an MBR and an ellipsoid.

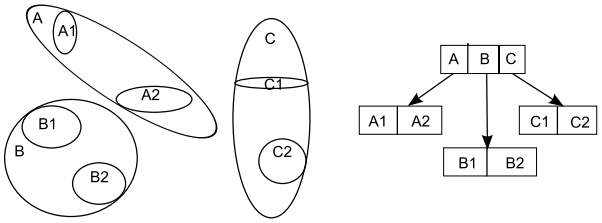


Figure 3.2: The eR-tree composed of ellipsoids.

3.2.1 Variant #1

The first idea of how to define eR-tree is apparently by the use of ellipsoids in leaf and also by inner nodes as it is shown in Figure 3.2. However, this approach turned out to be inappropriate because of the two reasons:

1. Inadequate complexity of Ellipsoid–Box intersection test. See section 4.4.3 and 5.2.2.
2. Inadequate complexity of ellipsoid "update". See step 7 in the algorithm 1.

The first problem arises when evaluating range queries. Providing that the tree is overlap free and the query returns some hits, we would need to test for Ellipsoid–Box intersection in average

$$\frac{node\ fanout}{2}\ tree\ height$$

times, where the node fanout usually ranges from 50 to 200. This inefficiency would definitely overcome any positive contributions ellipsoid can bring to

the eR-tree structure.
The second problem would arise during the construction time. While this is not crucial because we can afford some inefficiency during the construction in favor of search performance, the first problem is simply "no go" and using ellipsoids in inner nodes is no-through way.

3.2.2 Variant #2

To address the issues mentioned above and because most of the filtration in the R-tree is done in nodes just above the leaf nodes[1] we decided to investigate the variant with ellipsoids only in the pre-leaf nodes. All other nodes will use MBRs as regions. Consider figure 3.3.

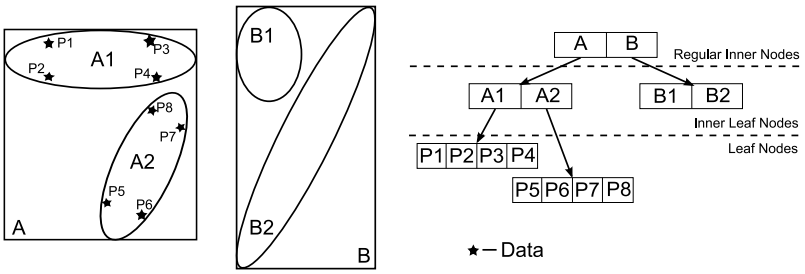


Figure 3.3: The eR-tree composed of ellipsoids and MBRs.

Even though ellipsoids in average cover the points more tightly, there are some situations when MBRs are superior to them. Basically in the following situations:

1. when splitting – ellipsoids tend to produce more overlaps then MBRs
2. when data are clustered in "rectangular" clusters – ellipsoids would cover more of the dead space

Observing Figure 3.4, we can notice the effects stated above – unnecessary death space coverage on the left and after splitting overlap between the ellipsoids on the right.

[1] We refer to them as *Pre-Leaf Nodes*

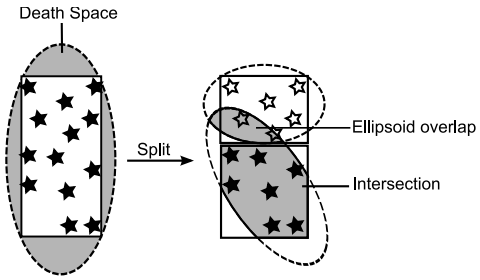


Figure 3.4: Death space coverage and overlapping split with ellipsoids.

3.2.3 Variant #3

To deal with the inefficiencies of the Variant #2 we decided to investigate the variant where the regions of pre-leaf nodes are defined as an intersection of ellipsoid and MBR (revisit Figure 3.4). All other inner nodes will use just MBRs. Adding extra MBR to the pre-leaf nodes will not dramatically decrease the node fanout. Experiments will confirm our presumptions.

3.3 Insertion

The algorithm 1 of the new entry insertion into the R-tree uses the procedure *Search(P)* to locate a leaf node which will hold the new entry. The pseudo code of our *Search(P)* procedure is provided in Algorithm 2.

Algorithm 2 Search(P)

Require: P – point to be inserted

1: $CurrentNode = RootNode$
2: **while** $CurrentNode$ is ***not*** leaf node **do**
3: **if** $CurrentNode$ is ***pre-leaf*** node **then**
4: $CurrentNode = getSubBranchEll(P)$ {return the child of the current node, which region is closest to the P. The distance between the ellipsoid and P is considered. See 4.4.1}
5: **else**
6: $CurrentNode = getSubBranchBox(P)$ {returns the child of current node, which MBR region being enlarged by P produces smallest enlargement. If the enlargement should cause an overlap with other regions of the CurrentNode and there exists the region of the CurrentNode which wont produce an overlap after being enlarged by P, then the sub-branch of this non-overlap-producing region would be chosen. I.e. we prefer rather to cover more of the dead space, than to produce the overlaps.}
7: **end if**
8: **end while**
9: **return** $CurrentNode$

3.4 Inner Node Splitting

Inner node splitting is taken from the ATOM[Gro03] R-tree implementation. The idea is to split an overfulled inner node into preferably same size halves, while minimizing the overlap and the dead space coverage. The overlap minimization is superior to dead space covering minimization.

3.5 Leaf Node Splitting

Leaf node splitting strategy significantly determines the performance of R-tree structure. Our splitting algorithm which tries to minimize variance is presented in 3. In the first phase, we choose the dimension in which data are most spread. Then we sort data according to this dimension and find the split position.

Algorithm 3 MinVar Split

Require: $\{p_1, \ldots, p_k\}$ set of entries to be split

1: $maxVar = 0.0$
2: **for** $i = 0$ to $dimension$ **do**
3: $var = computeVarianceInDimension(i, \{p_1, \ldots, p_k\})$
4: **if** $var > maxVar$ **then**
5: $maxVar = var$
6: $splitDimension = i$
7: **end if**
8: **end for**
9: $\{p_1^{'}, \ldots, p_k^{'}\} = sortByDimension(splitDimension, \{p_1, \ldots, p_k\})$
{so that $\{p_1^{'} \leq p_2^{'} \leq \ldots \leq p_k^{'}\}$ holds in $splitDimension$}
10: $diff = p_k^{'}[splitDimenstion] - p_1^{'}[splitDimenstion]$
11: **for** $i = 2$ to $k - 1$ **do**
12: **if** $p_1^{'}[splitDimension] + diff/2 < p_i^{'}[splitDimension]$ **then**
13: $splitOrder = i$
14: $break$
15: **end if**
16: **end for**
17: **return** $\{p_1^{'}, \ldots, p_{splitOrder}^{'}\}, \{p_{splitOrder+1}^{'}, \ldots, p_k^{'}\}$

Chapter 4

Ellipsoid Theory

4.1 Ellipsoid Definition

4.1.1 Expressing a Quadratic Form in *n* Variables Using Matrix Notation

Quadratic Form is polynomial of degree two in n variables.

$$F(x_1, x_2, x_3) = ax_1^2 + bx_2^2 + cx_3^2 + dx_1x_2 + ex_1x_3 + fx_2x_3$$

is an example of quadratic form in 3 variables. The example could be written in form

$$Q(x) = \sum_{i=1}^{3} q_{ii}x_i^2 + \sum_{i=1}^{3}\sum_{j=i+1}^{3} q_{ij}x_ix_j \qquad x_i \in R^3$$

where q_{ij} are the coefficients from the quadratic form. For arbitrary n we can define matrix $D = (d_{ij})$ in the following terms

$$d_{ii} = q_{ii} \qquad i = 1, \ldots, n$$

$$d_{ij} + d_{ji} = q_{ji} \qquad i, j = 1 \ldots n,\ j > i$$

for some arbitrarily choosen d_{ij}. Representing variables $x_1 \dots x_n$ as vector $\vec{x} \in R^n$ we can write

$$\begin{aligned} x^T Dx &= x_1(d_{11}x_1 + \ldots + d_{1n}x_n) + \ldots + x_n(d_{n1}x_1 + \ldots + d_{nn}x_n) \\ &= \sum_{i=1}^{n} d_{ii}x_i^2 + \sum_{i=1}^{n}\sum_{j=i+1}^{n}(d_{ij} + d_{ji})x_i x_j \\ &= \sum_{i=1}^{n} q_{ii}x_i^2 + \sum_{i=1}^{n}\sum_{j=i+1}^{n} q_{ij}x_i x_j \\ &= Q(x) \end{aligned}$$

If we put $d_{ij} = d_{ji} = (1/2)q_i$ for $i, j = 1, \ldots, n, j > i$ than D is known as ***symmetric*** *coefficient matrix defining the quadratic form* Q(x). For further details and examples see eg. [Mur01].

4.1.2 Ellipsoid

n-dimensional *Ellipsoid* is a higher dimensional analogue of an ellipse.

Definition 4.1.1. *Ellipsoid $\varepsilon(c, Q)$ in R^n with the center in $c \in R^n$ and the shape matrix $Q \in R^{nxn}$ is set of points*

$$\epsilon(c, Q) = \{x \in R^n |\ (x - c)^T Q(x - c) \leq 1\}$$

where the shape matrix is a symmetric positive semidefinite coefficient matrix representing some quadratic form.

The volume of an ellipsoid $\varepsilon(c, Q)$ is given by a formula

Theorem 4.1.2. $Vol(\varepsilon) = \frac{\pi^{n/2}}{\Gamma(n/2+1)} \frac{1}{\sqrt{detQ}}$ *where* $\Gamma()$ *is a gamma function.*

Further details and references on the proof of the theorem can be found in [SM04].

4.2 Minimum Volume Covering Ellipsoid

Definition 4.2.1. *For a given set $S = \{x_1, \ldots, x_k\}$ of n-dimensional points we define* Minimum Volume Covering Ellipsoid *as any ellipsoid $\varepsilon(c, Q)$ for which holds*

$$\begin{aligned} \forall x \in S : (x - c)^T Q(x - c) \leq 1 \qquad & \textit{containment} \\ \varepsilon_1(c_1, Q_1), \forall x \in S : (x - c_1)^T Q_1(x - c_1) \leq 1 \Rightarrow & \\ Vol(\varepsilon_1) \geq Vol(\varepsilon) \qquad & \textit{min. volume} \end{aligned}$$

For a clarity we will stick to use an abbreviation **MVCE**.

Basically we know about three distinct approaches how to construct the MVCE. First one published in early 80's is based on eigenvalue decomposition and can by found in [Bar82]. Almost ten years later Welzl published [Wel91] an algorithm based on randomized iterative construction. Finally, Kchachiyan formulates the problem of computing MVCE as an optimization problem and using interior-point method was for the first time addressed in [KT93].

4.2.1 Iterative construction of MVCE

For a given set $S = \{x_1, \ldots, x_n\}$, $x_i \in R^n$ of points we will outline how to construct the MVCE. We will denote $mvce(S)$ as the MVCE of set S. First, we introduce intuitive lemma without proof. The proof can by found in [Wel91].

Lemma 4.2.2. *$mvce(S)$ is determined by $(n+3)n/2$ points from S.*

We should emphasize that those $(n+3)n/2$ points does not have to lie on the boundary of the ellipsoid. This would be true, if we were constructing e.g. minimum volume ball.

Algorithm 4 Construct mvce(S,R)

Require: S - set of points, $R = \emptyset$ - auxiliary set
1: **if** $S = \emptyset$ or $|R| = (n+3)n/2$ **then**
2: $mvce(R) := prim(R)$
3: **else**
4: choose random $p \in S$;
5: $D := construct\_mvce(S - \{p), R)$;
6: **if** $p \notin mvce(R)$ **then**
7: $mvce(R) := construct\_mvce(S - p, R \bigcup \{p\})$;
8: **end if**
9: **end if**
10: **return** $mvce(R)$;

In the algorithm 4 we are recursively locating $R \subseteq S$ of size $(n+3)n/2$, which will define $mvce(S)$. Basically, we start with $R = \emptyset$ which defines the "empty" ellipsoid and we are testing if all the points of the set S are included in this ellipsoid. If some point is not, it is clear, that this point should by in R and therefore we add it and recursively continue testing.

[Wel91] provided some heuristics how to choose a random point in the line 4 of this algorithm. Using those heuristics overall computing time in the experimental results were improved by factor of 50 for 10 dimensional ellipsoids of 5000 points.

Expected complexity of algorithm 4 is *linear* in number of the points, exactly $O(\delta\delta!m)$ where $\delta = (n+3)n/2$ and $m = |S|$.

The only disadvantage of this algorithm is requirement of $prim(R)$ in the line 2 which computes $mvce(R)$ given determining $(n+3)n/2$ points. We were unable to identify suitable algorithm which would compute $mvce(R)$ for points in different dimensions (required dimensions were $2, \ldots, 15$). [Wel91] is not specific on any of these primitives, the experiment provided in [Wel91] were done in dimension $n = 10$. In addition, fast primitives for $n = 2$ can be found in [GS97].

4.2.2 Optimization construction of MVCE

First note on solving construction of the MVCE as an optimization problem using interior-point method were researched in [KT93]. The complexity of that proposal was afterwards improved in [TY07, SM04, KY05]. The later brigs also as a byproduct *Core Sets*[1]. In this paper we will stick to the MVCE construction described in [Mos05].

We want to minimize the volume (see 4.1.2) of the resulting ellipsoid, while the ellipsoid contains all points from S. Hence, the formulation of MVCE is following

$$\begin{aligned} \text{minimize} \quad & det(Q^{-1}) \\ \text{subject to} \quad & (x_i - c)^T Q (x_i - c) \leq 1 \qquad i = 1, \ldots, |S| \\ & Q \succ 0 \end{aligned} \tag{4.1}$$

This problem is not a convex optimization problem and by substitution of $A = Q^{1/2}, b = Q^{1/2}c$ can be written as

$$\begin{aligned} \text{minimize} \quad & log(det(A^{-1})) \\ \text{subject to} \quad & \|Ax_i - b\| \leq 1 \qquad i = 1, \ldots, |S| \\ & A \succ 0 \end{aligned} \tag{4.2}$$

[1] Core set is a small subset of the input points whose covering is "almost" same as the covering of the entire input, hence it can be used to optimize on large set of points.

what is a convex optimization problem which is unfortunately still difficult to solve. Luckily, the dual problem seems to be much easier. Consult exceptional [Boy04], especially chapter dedicated to duality. We will need to define *lifting* of the original problem. This mean, we will move all points $S = \{x_i, \ldots, x_k\}$ $x_i \in R^n$ to R^{n+1}. We can set $(x_i^l)^T = [x_i^T, 1]$ $i = 1, \ldots, k$ and define $S^l = \{\pm x_1^l, \ldots, \pm x_k^l\}$. The $mvce(S^l)$ will be symmetric about the origin of R^{n+1} and the $mvce(S)$ will by obtained as an intersection of the $mvce(S^l)$ with hyperplane $H = \{(x, 1) \in R^{n+1} | x \in R^n\}$. The lifted optimization becomes

$$\begin{array}{lll} \text{minimize} & log(det(M^{-1})) & \\ \text{subject to} & (x_i^l)^T M x_i^l \leq 1 & i = 1, \ldots, |S| \\ & M \succ 0 & \end{array} \tag{4.3}$$

where M is the decision variable. Now we can formulate and optimize a Lagrangian dual problem to the lifted problem. The optimization will be carried out by Conditional Gradient Ascent method. The Gradient Ascent/Descent method can be simplified as

Algorithm 5 Ascent Gradient Method

1: **while** stopping criterion not satisfied **do**
2: Compute Ascent Direction {*which way to approximate solution*}
3: Line Search {*compute the length of step*}
4: Update {*update the approximation*}
5: **end while**
6: **return** $mvce(R)$;

Asymptotic complexity of this algorithm is linear in number of points being covered by ellipsoid. For further details, consult [Mos05] and [KY05]. In former we can find exact formulas how to compute Ascent Direct, Step Size or extract the original solution from the solution of the dual problem. In later is derived asymptotic complexity. Generally useful reading could be [Boy04], especially chapter 9.

As a stopping criterion is used an average distance of all points which lies outside the approximated ellipsoid to its boundary. While we need for the purposes of this work to be all points strictly included in the resulting ellipsoid, we are performing a post processing on the ellipsoid obtained from the approximation. In this post processing we locate the point which lies furthest from the boundary of the ellipsoid and than we scale the ellipsoid so this point(and all other) lies inside.

We conducted some small performance experiments to evaluate an adequacy of this constructing technique of the MVCE for our purposes. In 4.1 we can observe what is the impact of the size of S and dimension (2, 5, 10, 15, 20) on the time needed to construct the MVCE. The stopping criterion of the algorithm was set to 0.01. Experiments were done in a matlab and were carried out on 2.2GHz AMD Turion processor. Exact construction times are included in figure 4.1 to provide better image of effectiveness. Other

SetSize	2	5	10	15	20
10	0.003	0.002	0.000	0.002	0.000
20	0.009	0.008	0.002	0.002	0.002
40	0.008	0.006	0.008	0.005	0.002
60	0.016	0.013	0.017	0.003	0.008
80	0.019	0.020	0.017	0.013	0.011
100	0.022	0.025	0.028	0.023	0.019
140	0.036	0.051	0.052	0.055	0.033
160	0.047	0.054	0.094	0.072	0.053
200	0.069	0.081	0.155	0.117	0.094
220	0.089	0.103	0.184	0.148	0.114
250	0.119	0.105	0.156	0.131	0.125
280	0.334	0.345	0.388	0.325	0.250
300	0.366	0.411	0.437	0.383	0.267
350	0.522	0.555	0.642	0.559	0.411
400	0.701	0.769	0.800	0.705	0.578
450	0.892	0.919	1.059	0.970	0.781
500	1.133	1.167	1.295	1.252	1.012

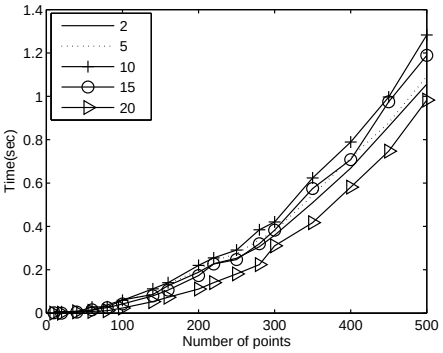


Figure 4.1: The MVCE construction time as parameter of $|S|$ and the dim.

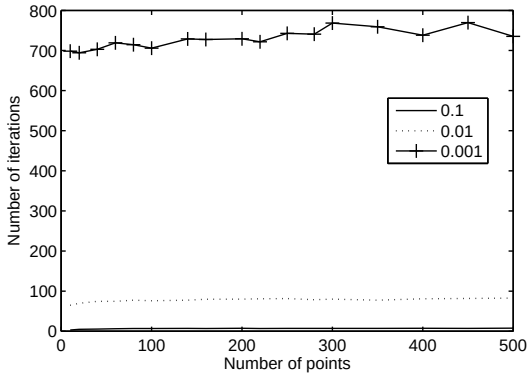
experiments were focused on the stopping criterion. In 4.3 we can notice that the efficiency of the algorithm mainly depends on this parameter. In 4.2 we can observe the number of iterations required till the stopping criterion is reached. These experiments were conducted with $n = 3$. This algorithm fulfills the requirements for our purposes with stopping criterion at most equal to 0.01.

4.3 Extracting ellipsoid properties

Eigenvectors of the shape matrix Q defines the direction of ellipsoid axes. The length of axes is given by

$$length(\lambda) = \frac{1}{\sqrt{(\lambda)}}$$

See picture 4.4.



SetSize	0.1	0.01	0.001
10	2.6	66.2	710.0
20	4.4	74.6	713.8
40	6.2	72.2	705.8
60	6.6	75.6	782.8
80	5.8	74.4	713.8
100	7.0	76.6	704.4
140	6.6	76.6	683.8
160	7.0	77.8	722.6
200	7.4	76.4	710.0
220	7.2	80.2	740.2
250	7.4	80.0	707.2
280	7.4	79.2	719.8
300	7.6	80.6	785.4
350	7.4	79.8	748.2
400	7.2	81.6	733.6
450	7.6	82.2	750.4
500	7.0	78.4	739.8

Figure 4.2: Number of iterations required to construc the MVCE.

4.4 Other operations

4.4.1 Point and ellipsoid distance

The distance of the point $p \in R^n$ and the ellipsoid $\varepsilon(c, Q)$ is defined as

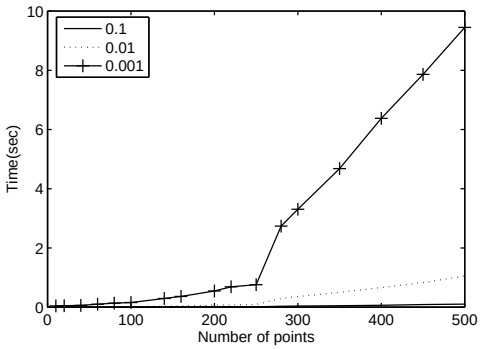
$$dist(p, \varepsilon) = (p - c)^T Q(p - c)$$

4.4.2 Hyperplane and ellipsoid distance

Given a *hyperplane* $H = \{x \in R^n | \ \langle l, x \rangle = \sigma\}$ where $l, \sigma \in R^n$ are fixed, the distance between H and $\varepsilon(c, Q)$ is given by

$$dist(H, \varepsilon) = \frac{|\sigma - \langle l, q \rangle| - \langle c, Q^{-1}c \rangle^T}{\langle c, c \rangle^{1/2}}$$

where $\langle \cdot \rangle$ denotes standard dot product. If $dist(H, \varepsilon)$ is negative, ellipsoid and hyperplane intersect.



(a)

SetSize	0.1	0.01	0.001
10	0.084	0.003	0.053
20	0.003	0.003	0.060
40	0.003	0.012	0.069
60	0.009	0.013	0.128
80	0.003	0.019	0.156
100	0.006	0.022	0.163
140	0.000	0.041	0.362
160	0.009	0.047	0.716
200	0.013	0.072	0.588
220	0.016	0.075	0.744
250	0.009	0.081	0.850
280	0.031	0.316	2.853
300	0.034	0.372	3.363
350	0.056	0.494	4.650
400	0.062	0.634	6.613
450	0.078	0.850	8.169
500	0.103	1.044	10.013

(b)

Figure 4.3: The MVCE construction time as parameter of $|S|$ and tolerance.

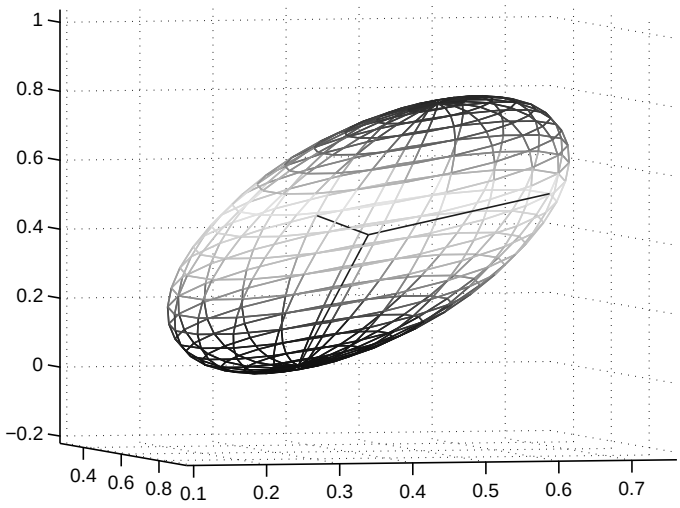


Figure 4.4: 3D ellipsoid with highlighted axes.

4.4.3 Testing for box and ellipsoid intersection

The problem of deciding whether box $B(ql, qh) \quad ql, qh \in R^n$ and ellipsoid $\varepsilon(c, Q)$ intersect can be formalized as a convex optimization problem of a form

$$\begin{aligned} \text{minimize} \quad & (x-c)^T Q(x-c) \\ \text{subject to} \quad & ql \leq x_i \leq qh_i \qquad i = 1, \ldots, n \end{aligned} \tag{4.4}$$

where objective variable is x. If $x \leq 1$ than the box and the ellipsoid intersects, otherwise their intersection is empty. Some heuristics can be applied here so under some circumstance optimization is not necessary. For example we can test weather ellipsoid intersect any plane defined by the faces of the the box using 4.4.2. If none intersects, neither ellipsoid intersects with the box.

4.4.4 Bounding ellipsoids by ellipsoid

For completeness we provide reference to [Yil06] where is discussed an algorithm for computing minimum volume bounding ellipsoid of ellipsoids. The running time is asymptotically linear in number of ellipsoids being bounded.

Chapter 5

Implementation

In the beginning of this chapter, we will introduce a concept of the framework used to implement tree-like index structures. Later on, we will briefly discuss existing R-tree implementation based on this framework, which we had modified to comply with our eR-tree proposal. Finally, in the rest of the chapter we will focus on the details of ellipsoid class implementation.

At this point, we would like to emphasize that we are aware of some implementation inefficiencies, which we do not bother to eliminate because the aim of this work is not to implement highly optimized solution, but rather thoroughly investigate properties of an ellipsoid in the R-tree index structure.
Anyway, along the text we will point out where further code optimization, we believe, can be applied. Also we will try to "forecast" to what extend implementation inefficiencies can effect the experimental results given in Chapter 6.

5.1 Amphora Three Object Model

ATOM(**A**mphora **T**hree **O**bject **M**odel) is a C++ object framework for developing and testing tree-like index structures. It was developed at VŠB - Technical University of Ostrava to support activities of Amphora Research Group.

The main characteristics of ATOM can be summarized as

- heavy use of C++ templates

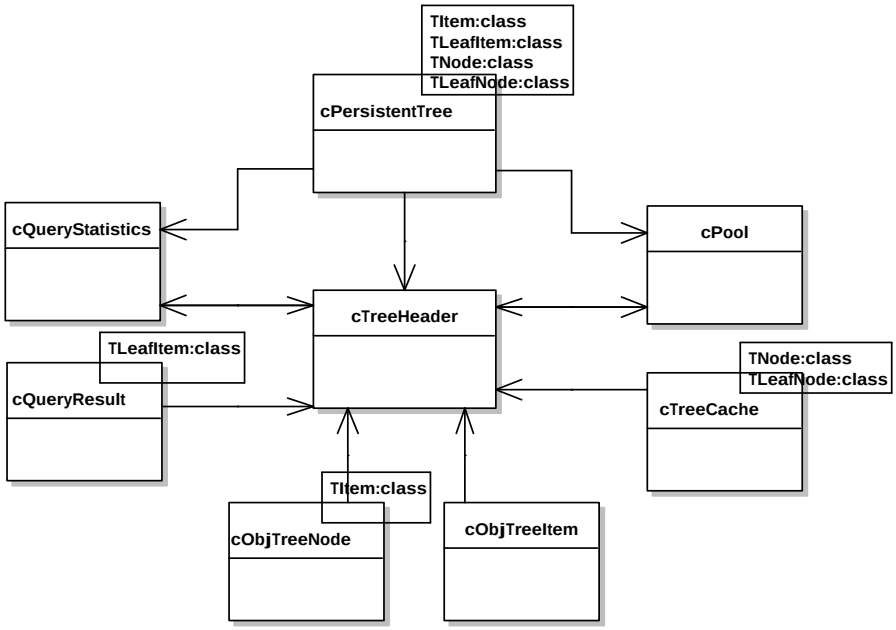


Figure 5.1: ATOM class diagram.

- optimization of access to secondary storage – caching
- RAM-memory access optimization – pooling
- heavy optimizations on code-level

5.1.1 ATOM Design

The basics of ATOM design are depicted in the Class diagram presented in Figure 5.1. In following, we will provide short description of the main building blocks.

cTreeHeader class is a "configuration" class which specifies rudimentary parameters of the tree structure like is

- mNodeItemCapacity is the number of items inner nodes can hold. In R-tree implementation, item represents MBR or other types of regions.
- mLeafNodeItemCapacity is the number of items leaf nodes can hold. While inner nodes hold, apart from items, only references to child nodes, leaf nodes can hold actual data. Generally, the size of inner

nodes and leaf nodes is the same and is aligned to external storage block size. And because the size of pointer to child node and the size of actual data can vary, we need to be able to have different capacities of the inner and the leaf nodes.

- mNodeFanoutCapacity is the number of pointers to child nodes inner node can hold. In R-tree index structures this is typically equal to mNodeItemCapacity. The same parameter also exists for the leaf nodes.
- mNodeRealSize holds the real size of the inner node (not aligned to an external storage block size). The same parameter else exists for the leaf nodes.

cTreeHeader holds also some counters like is

- mHeight – height of the tree
- mInnerNodeCount – number of the inner nodes
- mLeafNodeCount – number of the leaf nodes
- mInnerItemCount – total number of the items in all inner nodes
- mLeafItemCount – total number of the items in all leaf nodes

cTreeHeader apart from configuration parameters and different counters holds also references to other tree-specific classes like is a cache or pool. Classes of ATOM usually access other classes just through the references in cTreeHeader.
The concrete tree structures should create a specification of cTreeHeader class where additional parameters like dimension or cardinality can be provided.

cPersistentTree class is a core abstract implementation of the tree structure. It provides a fundamental methods to Create() new index structure or to Open() existing. It manages memory allocation and initialization of the cache and the pool. It is parametrized by TNode, TLeafNode, TItem and TLeafItem.

ObjTreeItem class is an abstract implementation of the "TItem" interface. Objects of this class represents regions.

cObjTreeNode class is an abstract implementation of "TNode" interface and is parametrized by the TItem. Objects of this class are nodes of the tree. mLeaf flag is used to distinguish the inner nodes from the leaf nodes (inner and leaf nodes are also differentiated by a indexId, more

on this later in this chapter). Most important method, apart from the obvious like is a method to add item etc., is methods to **Split()** node when the node is overfull.

cPool class together with the cTreePool class provides pools to avoid extensive memory allocations on heap during indexing or searching. cPool holds object of a generic types like are integers, arrays of integers, etc. cTreePool holds tree specific object like Nodes, Items and arrays of these.

cTreeCache class provides caching of nodes. This class has separate methods for reading or writing of the inner nodes and for reading and writing of the leaf nodes. This way inner and leaf nodes can be cached separately (usually more space is dedicated to cache the inner nodes). For caching are used two-dimensional arrays with chained hashing. The size of the cache is given when the index structure is created or opened. cTreeCache has also methods that provides statistics on hits into the cache and number of real writes or reads.

cQueryResult class is a container for a search results.

cQueryStatistics class holds various counters that can by useful when evaluating the performance of an index structure.

More details can be found in [Gro03]. The document is written in Czech language.

5.1.2 R-tree implementation

In this section, we will give a short description of the R-tree implementation from which we derived our solution. We will highlight the most important methods and concepts so the reader can easily browse through the code. Please regard Figure 5.2.

cCommonRTree.* classes contain code, which is common for all R-tree variants.

cCommonRTree class contains method for the tree initialization, closing and debugging but mainly:

- Insert(TLeafItem &item) insertion algorithm (See Algorithm 1) implementation

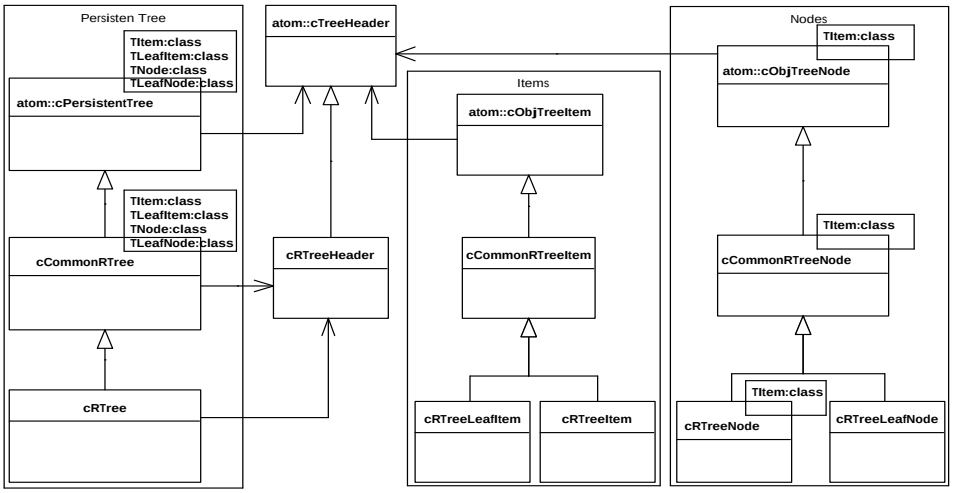


Figure 5.2: Class diagram of ATOM R-tree implementation.

- RangeQuery(cTuple &ql, cTuple &qh) range query implementation

cRTreeNode class

- Split(cRTreeNode &newNode) this method is called, when over-fulled inner node need to be split (I.e subroutine SplitInnerNode(InnerNode) in insertion Algorithm 1).
- FindMbr(cTuple &tuple, int &itemOrder) this method is called in the insertion Algorithm 1 in the step 1 to find suitable branch of the tree for insertion of the new entry(tuple).
- FindNextRelevantMbr(int currentOrder, cTuple &ql, cTuple &qh) is a method used in the search algorithm to choose which branches of the tree intersect with query region and therefore need to be searched.

cRTreeLeafNode contains merely method to Split(cRTreeLeafNode &newNode) over-fulled leaf node (represent SplitLeafNode(LeafNode) routine in the insertion Algorithm 1).

cRTreeItem is a simple MBR specified by two tuples (n-dimensional points).

5.1.3 Modification

Event thought ATOM tries to be as versatile as possible, we won't avoid some modifications of this framework, which are necessary for eR-tree imple-

mentation. In particular, we need to distinguish in addition to the obvious leaf and inner nodes also the *pre-leaf* nodes.
Pre-leaf nodes are nodes of the tree on level 0 which are nodes just above the leaf nodes. See Figure 2.2, pre-leaf nodes are nodes on level labeled as (L0).

The reason to distinguish pre-leaf nodes is that these nodes will hold items that specifies region as an intersection of an ellipsoid and a MBR (Revisit the eR-tree proposal, varian #3). Taking in account that the MBR item require less space to be stored than ellipsoid and because all nodes of tree ought to be of the same size it is necessary to adjust the fanout of pre-leaf nodes.

To distinguish pre-leaf nodes we used same attributes that are used to distinguish inner and leaf nodes apart and that is:

- node index
- flag

Because ATOM keeps inner and leaf nodes in different caches, it is desirable to know before reading the node (having just the node index) what type of the node it is. Therefore, the index of the leaf node has the most significant bit turned on. The same approach is applied to the pre-leaf nodes. If the second most significant bit of index is set, then we know, we are dealing with the pre-leaf node.

The type of existing node is set with a method SetIsPreLeafNode(bool tf).

cTreeHeader class is extended with attributes specifying parameter of the pre-leaf nodes and counter connected with them:

mPreLeafNodeRealSize is the real size in bytes of the pre-leaf node (not aligned).

mPreLeafNodeCount is the number of the pre-leaf nodes in the tree.

mPreLeafNodeItemSize is the size of one item, which the pre-leaf node hold.

mPreLeafNodeFanoutCapacity is the number of pointers to the child node (in this case the leaf nodes) pre-leaf node can hold at most.

mPreLeafNodeItemCapacity is the number of items pre-leaf node can hold at most. It is equal to mPreLeafNodeFanoutCapacity in case of the eR-tree.

We also modified all concerned methods like is e.g. ComputeNodeCapacity() to reflect the changes.

5.2 Ellipsoid class

Ellipsoid primitives are implemented in the cEllipsoid class.

We decided to use newmat10[1] matrix library because it

- Supports all the required operations - Matrix Inverse, Cholesky, SVD, Eigenvalue decomposition.
- Is designated for larger matrices (when we want to bound e.g. three hundred 5-dimensional points we need matrix of dimension 300x6).
- Provides overloaded basic matrix operations.
- Is Straightforward in design.
- Provides lazy evaluation.

This library is rather rich in functionality, than thoroughly optimized. If further optimization of eR-tree implementation would be required, this could be the starting point. During ellipsoid manipulation, we mostly deal with symmetric matrices that can be also markedly optimized.

cEllipsoid class provides following methods:

boundTuples(cTuple *tuples[], int tupleCount) constructs the MVCE which bounds given tuples, see 5.2.1 for an implementation details.

isTupleContained(cTuple &tuple) returns true, if tuple is contained in this ellipsoid.

tupleDistance(cTuple &tuple) returns the distance of tuple from this ellipsoid. See definition 4.4.1.

testBoxIntersection(cTuple &ql, cTuple &qh) returns true, if the box defined by ql and qh (lower and higher corners) intersects with this ellipsoid. See 5.2.2 for implementation details.

getAbsoluteVolume() returns the absolute volume of this ellipsoid. See Theorem 4.1.2.

boundByBox(cTuple &ql, cTuple &qh) bounds this ellipsoid with a box. Upper and lower box corner are returned in method arguments. See 5.2.3 for implementation details.

[1]The package can be obtained at http://www.robertnz.net/nm10.htm.

5.2.1 Constructing the MVCE

Algorithm described in 4.2.2 is used to construct the MVCE.

```
//cEllipsoid.h

#define APPROXIMATION_DEFECT 0.1
void boundTuples(cTuple *tuples[], int tupleCount);
```

Approximation_Defect macro is used to define the tolerance (stopping criterion) of the approximation. When constructing the MVCE we must threat some cases of an input in special way. The MVCE construction algorithm will produce either degenerated ellipsoid or will crash because of matrix singularity in two following situations:

- When we try to bound "insufficient" number of points. As an example can serve bounding of 1 point in 3-dimensional space. Resulting ellipsoid should be a "point" ellipsoid, what will (taking in account floating-point arithmetic) result in NaN run-time errors.
- When we try to bound points which lies (or nearly lies) on line segment or plane etc. In such situation, we can expect same behavior as in previous.

First problem is resolved by bounding less points then is the dimension by a sphere with appropriate radius(sphere is also ellipsoid, so we represent it in the same fashion with shape matrix and center).
To avoid second problem, we are adding extra points to the input of approximation, so the points don't lie on the line, plane etc.

Ellipsoid shape and ellipsoid center are stored in ColumnVector ellipsoidCenter_ and SymmetricMatrix ellipsoidShape_.

5.2.2 Testing for box and ellipsoid intersection

The implementation of box and ellipsoid intersection is based on 4.4.3.

We decided to use loqo[2] solver to solve the optimization problem. The reasons of this choice are mostly the simplicity and effectiveness of the package.

To properly initialize loqo solver we need to provide following methods:

[2]Available on http://www.princeton.edu/~rvdb/loqo/install.html

- double objval(double *x) – returns the value of the objective function in x
- void objgrad(double *c, double *x) – returns through c the gradient of the objective function at x
- void hessian(double *Q, double *x, double *y) – returns through Q the hessian of the objective function
- int stop_rule(void *vlp) – this method decides whether to carry on with optimization or to stop. The stopping rules are following:
  - $opt\_value \leq 1$. If this stopping criterion is reached, ellipsoid and box intersects.
  - $abs(opt\_value - opt\_value\_previous\_iteration) \leq 1.0e-5$. We stop if we are "close enough" to optimal solution.
  - iteration limit reached. Max number of iterations is set to 60. If this limit is reached we proclaim that box and ellipsoid intersects (to avoid false dismissal).

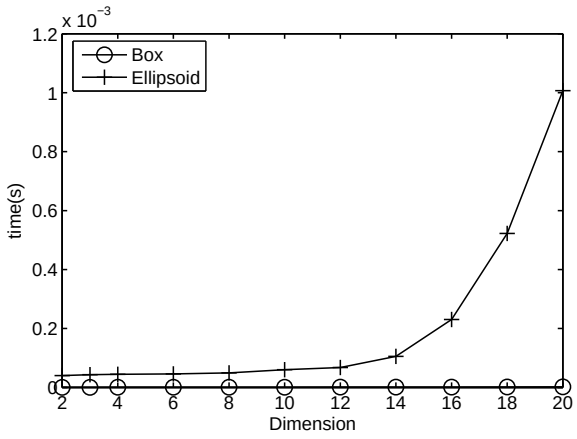
Loqo solver is initialized as static variable. The class cEllipsoid provide method testBoxIntersection(cTuple &ql, cTuple &qh) which initialize the problem and then solve it. Other options set to speed up the optimization are:

- convex=1 – assert the problem is convex thus disabling special treatment required by non-convex problems
- quadratic=1 – assert the problem is quadratic
- lincons=1 – assert the problem has only linear constraints

As a starting point of optimization is given the center of the box.

For further details please consult [Van06].

We have performed a performance tests to compare the efficiency of the ellipsoid–box vs the box–box intersection testing which are presented in Figure 5.3. The performance test were carried out on 2,2 GHz AMD Turion, the problem was implemented in C++. Test consisted of 50 iterations. In each iteration, we randomly generated one ellipsoid/box and tested it for intersection with 5000 other randomly generated boxes. In Figure 5.3 are presented measured times recalculated for one ellipsoid/box–box intersection test. We can observe, that the Ellipsoid–Box intersection test is much more expensive in terms of CPU time than Box–Box intersection test.



(a)

Dimension	Box	Ellipsoid
2	3.12e-7	4.03e-5
3	3.74e-7	4.34e-5
4	5.00e-7	4.45e-5
6	8.12e-7	4.58e-5
8	6.86e-7	4.93e-5
10	8.12e-7	6.02e-5
12	1.12e-6	6.76e-5
14	1.19e-6	1.06e-4
16	1.20e-6	2.30e-4
18	1.37e-6	5.22e-4
20	1.94e-6	1.01e-3

(b)

Figure 5.3: Comparison of the Ellipsoid–Box and the Box–Box intersection test.

5.2.3 Bounding Ellipsoid by a Box

The algorithm to bound ellipsoid by a box can be considered as naïve, but for our purposes seems to be sufficient. In future, here can by applied further optimization.

The algorithm to compute bounding box of ellipsoid is as follows:

Algorithm 6 Bound an ellipsoid by a box.

Require: ε(c, ellipsoidShape)

```
[V D] = eig(ellipsoidShape) {eigenvalue decomposition}
vertexes = computeEllipsoidVertexes(V,D) {compute all vertexes of ellipsoid}
ql = min(vertexes)
qh = max(vertexes)
isEllipsoidBounded = test(ql,qh) {test, whether ellipsoid is bounded}
while !isEllipsoidBounded do
    ql = ql - approximationDefect_
    qh = qh + approximationDefect_
    isEllipsoidBounded = test(ql,qh)
end while
return
```

To test, whether ellipsoid is bounded by box, distance of all the box faces

from ellipsoid are computed (see 4.4.2).

Chapter 6

Experimental Results

In this section, we give our experimental results. First, we provide a characteristics of data sets used to benchmark the eR-tree. Then we describe the indexing performance together with parameters measured. Finally, we describe the querying performance. As a result, we aim to classify which data is suitable for the eR-tree.

6.1 Notes on a measurements

6.1.1 Measurement settings

All experiments were performed on the 2.2 GHz AMD Turion processor, 1GB RAM, with 5400-rpm hard drive. C++ source code was compiled with VC 8.0 in the windows XP environment (fs=NTFS).

6.1.2 Measured Parameters

I/O The I/O cost is the most meaningful indicator of the index structure performance. Because in practice greater deal of the cache is dedicated to the inner nodes compared to the leaf nodes (often all inner nodes are hold in a RAM), we will *separately* measure the I/O operations for the inner nodes and the leaf nodes, so it can be estimated what can be the expected I/O savings when placing inner nodes into the RAM.
ATOM provides one 2-dimensional[h,w] cache table for the inner nodes and one (of the same structure) for the leaf nodes (position of the node

in table is node_index%h, then **LRU** stategy is used to place colliding nodes in w positions). We will separately measure real I/O (actual access to secondary storage) and I/O answered by the cache. In graphs, we will stick to present sum of real I/O and cache hits so the results are not confinguration dependent (how much memory is allocated to cache). If the reader is interested in the number of real I/O and number of cache hits, we advice him to examine log files. See B for organization of attached compact disk with log files. During the tests, the cache of default size 20x2 will be used, if not otherwise stated.

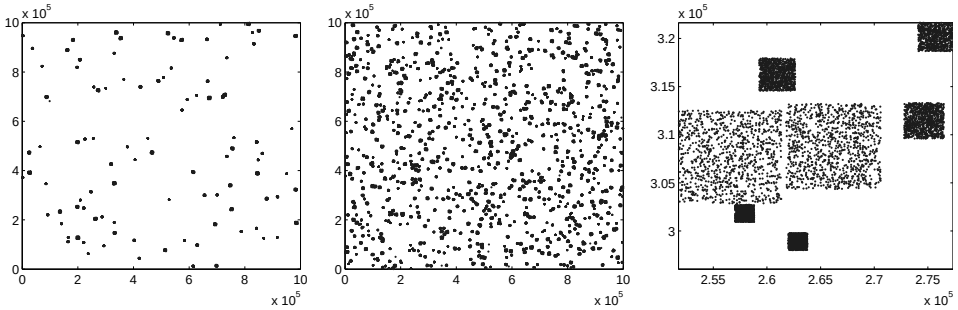
Execution Time We will present execution time in test results, however we believe this is the least significant attribute describing structure performance. As "Execution Time" will be presented *process user* time.

6.1.3 Opposing R-tree

For a comparison, results for eR-tree are presented along with results for R-tree. Opposing R-tree is original Guttman's version [Gut84] with small modifications. These modifications try to avoid regions overlaps at the cost of higher dead space coverage. It is discussed in [BKSS90] that regions overlaps have worse impact on the search performance than the dead space coverage; therefore, these modifications should slightly improve the performance of the original R-tree.

6.2 Data Sets

Here we describe data sets used to benchmark the eR-tree. Notice, that we did not tested eR-tree on *non-clustered* data with Gaussian (or uniform) distribution as is usual, because it is clear that the eR-tree will be outperformed by any variation of the R-tree which uses *MBRs* (MBRs tend to produce much less overlaps than ellipsoids and with non-clustered Gaussian data overlapings becomes barely the only determinating factor of the performance). However, without giving any further details, the eR-tree on non-clustered Gaussian data required in average 15% more I/O than the R-tree in our perfunctory tests.



(a) $1x10^5$ point data set. (b) $1x10^6$ point data set. (c) Zoomed clusters.

Figure 6.1: 2-dimensional synthetic clustered data.

6.2.1 Synthetic Clustered data

To approximate real data, we have generated *clustered* data with uniform distribution inside the clusters. Data sets with sizes $1x10^4$, $4x10^4$, $8x10^4$, $1x10^5$, $5x10^5$, $1x10^6$, $2x10^6$ were generated[1]. The domain size is $1x10^6$. One cluster contains 1000 points and is generated with rand((0, 1 >)*10000 spread. Two-dimensional data are presented in Figure 6.1. Individual records were randomized prior storing (so they are not indexed "cluster by cluster" but randomly).

6.2.2 Real IMDB data

To evaluate the eR-tree on real data we reached for the well-known movie database IMDB[2]. Because the database is distributed as separate text files (actors.txt, titles.txt, ...), first we converted data into SQL database using imdbpy[3] project. As a byproduct of the conversion all required text elements were assigned with a numerical value (primary keys of relevant tables) which are indexed (eR-tree implementation can index only numerical data, not lexical). Afterward we generated desired data set with select statement:

```
SELECT Title.id, Cast.person_id, Info.genre, Title.production_year,
       Title.kind_id, FROM Title
  INNER JOIN cast_info Cast ON
                       Cast.movie_id = Title.id AND
                       C.role_id = 8 //only directors
```

[1] In further text we will refer to this data sets as *SCU1x4*, *SCU4x4*, etc.
[2] Available at http://www.imdb.com/
[3] http://imdbpy.sourceforge.net/

Attribute	MIN	MAX
Title.id	1	1137185
Cast.person_id	31	1824450
Info.genre	2	29
Title.production_year	1519	2013
Title.kind_id	1	7

Figure 6.2: Cardinality of individual attributes of the IMDB data set.

```
INNER JOIN movie_info Info ON
                      Info.movie_id = Title.id AND
                      Info.info_type_id = 3 //genre id
```

The semantics of the attributes in select statement is summarized below:

Title.id is an id of the movie. Each movie can appear in result set only once.

Cast.person_id is an id of the movie director. If the movie has been directed by multiple directors, the first one is chosen.

Info.genre is an id of the movie genre. If the movie was tagged with multiple genres (war, drama, ...) the first genre was chosen.

Title.production_year is a production year of the movie.

Title.kind_id is an id of the movie "kind" (evening movie, short movie, serie, ...)

Each record of the result set represents point in 5 dimensional space. Entire result set consist of 392689 records. Cardinality of individual attributes can be found in Table 6.2.

6.2.3 Real Trajectory data

Trajectory data set consists of 276 trajectories of 50 trucks delivering goods in Athens city district for 33 days[4]. Each record consist of x_pos, y_pos, date (unix time), truck_id., X and Y coordinates are in GGRS87 reference system. Trajectory data set consist of 112203 individual records. Distribution of the positional coordinates can be found in Figure 6.3. Data is ordered by the time as they were captured, as they would be inserted into database in real-life.

[4]Available at http://www.rtreeportal.org/datasets/trajectories/trucks.zip

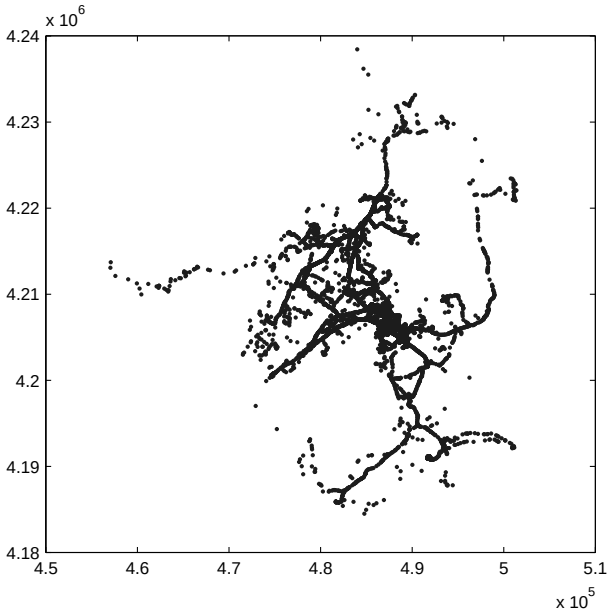


Figure 6.3: X,Y attributes of the trajectory data set.

6.3 Indexing

6.3.1 Index size and utilization

Here we compare the eR-tree with the R-tree in terms of the index size. All data sets were tested with dimension equal to 5 (except trajectory data set, which is comprised of only 4 dimensional entries). Both structures have node size set to 4096 KB (page aligned).
In Table 6.1 are provided measured values. On the left from the double vertical line are the results for the eR-tree; on the right are the results for the R-tree. Together with index sizes are provided counts of the inner nodes (#IN) and the leaf nodes (#LN). We can notice that on SCU data sets eR-tree reached better results than the R-tree in terms of the index size. The opposite holds for IMDB and TRAJ data sets. The question is, what is the reason of this behaviour? First, we will answer, weather the eR-tree performance deteriorates on real data sets or contrary the R-tree performance improves on real dataset.

To answer this question we will study space utilization of both trees. Space utilization represent in per cents how "good" is space being occupied by index utilized. If space utilization of the eR-tree with real data set will decrease, we will know, that eR-tree's performance is degrading on real datasets. Re-

Data Set	eR size	#IN	#LN	R size	#IN	#LN
SCU1x4	352	7	80	488	14	107
SCU4x4	1544	30	355	2244	147	413
SCU8x4	3260	76	738	4992	407	840
SCU1x5	3620	60	844	5232	265	2042
SCU5x5	22072	650	4867	29532	2064	5318
SCU1x6	43136	1194	9589	55080	3160	10609
SCU2x6	92369	3307	19791	110060	6439	21075
IMDB	18188	387	3959	15156	204	3584
TRAJ	4868	99	1117	3668	20	896

Table 6.1: Comparison of eR-tree and R-tree index sizes, in KB.

versely, if the R-tree utilization will increase with real data sets, we will know that R-tree's performance is improving.

We distinguish following types of space utilization:

Leaf nodes utilization(LeafU) express how many items of all leaf nodes items are actually occupied. Recall, that leaf nodes are comprised of items (same as inner nodes) and that the maximum number of items node can hold is called fanout. For example, having index that holds 400 entries and is comprised of 4 leaf nodes, where fanout of one leaf nodes is 150 we will get $400/(4*150)*100 = 67\%$ leaf nodes utilization. In brief, leaf node utilization represents how much are leaf nodes full.

Pre-leaf nodes utilization(PLeafU) represents utilization of the *pre-leaf* nodes. Because inner nodes and pre-leaf nodes have different fanout, we have to count them separately. This applies only to the eR-tree.

Inner nodes utilization(InnerU) is the same as leaf node utilization but for inner nodes. When computing utilization of eR-tree first we compute utilization of *inner nodes* and *pre-leaf nodes*. The resulting utilization is the average of those two utilizations.

Avarage utilization(AvgU) is average utilization of leaf nodes and inner nodes.

In Table 6.2 are presented the results. On the left from the doubled vertical line are results for the eR-tree, on the right for the R-tree. We can observe, that R-tree's performance is increasing on real data sets – ≈62% compared with ≈49% space utilization on SCU data sets. However, also eR-tree's

Data Set	AvgU	InnerU	LeafU	PLeafU	AvgU	InnerU	LeafU
SCU1x4	73	41.5	73.5	70	51.9	9.2	55
SCU4x4	66	59.6	66.3	64	48.4	4.1	57
SCU8x4	63.6	53.6	63.8	51.7	45	3.3	56
SCU1x5	69.7	74.4	69.7	75.2	50.2	5.3	56.5
SCU5x5	59.2	20.5	60.4	67.1	46.3	3.8	55.3
SCU1x6	60.7	30	61.3	49.7	48.3	4.7	55.4
SCU2x6	58.6	25.8	59.4	34.9	48.5	4.6	55.8
IMDB	57.8	33.1	58.3	37.7	63.1	20.0	64.5
TRAJ	49.2	45.7	49.2	43.8	61.1	40.5	61.4
ϕ	62	43	62	55	51	11	57

Table 6.2: Node utilization of the eR-tree and the R-tree.

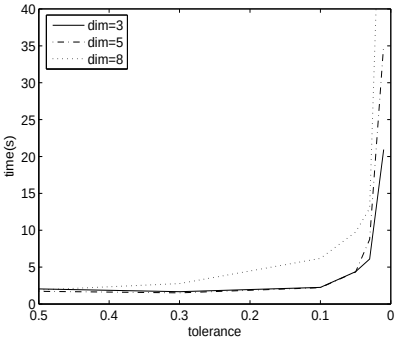
performance is degrading with real data sets – ≈53% compared with ≈64% space utilization on SCU data sets. In the talbe we can also notice very poor space utilization of the inner nodes of the R-tree.

6.3.2 Impact of a stopping criterion on indexing speed

Here we will discuss impact of a stopping criterion (more meaningful term would be *tolerance*) of the algorithm used to construct MVCE. Stopping criterion express how "good" the approximation of MVCE is, possible values are from interval (0,1). The scalar represent how big discrepancy between the approximation and "absolute" MVCE is bearable for us (stopping criterion equal to 0.001 will produce small discrepancies, on the other hand, stopping criterion equal to 0.9 will produce noticeable discrepancies). With stopping criterion, two important parameters in context of the eR-tree are determinated:

1. The indexing speed. Approximation that is more precise requires more iterations of the MVCE construction algorithm. We would like to index as fast as possible, so we would like the stopping criterion to approach to 1.

2. The dead space coverage. Approximation that is more precise will produce ellipsoids smaller in volume. Therefore, we would like to have stopping criterion approaching 0.



Tolerance	Dimension 3	5	8
0.5	2.04	1.71	1.90
0.3	1.67	1.51	2.76
0.1	2.25	2.21	6.18
0.05	4.3	4.34	9.75
0.03	6.09	8.75	12.96
0.01	20.9	35.15	72.56
R-tree	1.68	2.62	4.625

Figure 6.4: Stopping criterion (tolerance) as a parameter of indexing speed. Measured values are in seconds.

In Figure 6.4 we present the results of experiment, where we indexed SCU1x4 data set in different dimensions with different stopping criterions. In the graph, we can see the speed of indexing as a parameter of stopping criterion. In the table on the right are actual data used to generate the graph. In the bottommost line of the table is presented the time of the R-tree for comparison. In the table we can notice some inconsistencies (like indexing 3-dimensional data is faster with tolerance=0.3 compared to tolerance=0.5). Closer examination of the log file revealed, that this "inconsistency" is caused by a cache. This table gives us overview of the impact of stopping criterion on indexing speed – first parameter determinated by stopping criterion.

How is the second parameter affected by the stopping criterion? The answer can be found in Table 6.3. It is out of the scope of this work to compute exact intersecting volume of MBR and ellipsoid, therefore we will present volume of the pre-leaf nodes separately for ellipsoids and MBRs. In Table 6.3 we can see measured volume (E) for ellipsoids and (MBR) for MBRs. In the bottommost line are equivalent volume measurements for the R-tree.

We can observe that increasing accuracy of the approximation does not lead to significant reduction in volume, in contrast to indexing speed, where we could observe that higher accuracy require much more execution time. From this observation we can conclude, that the most suitable value for stopping criterion could be ≈ 0.1, which will be used in the rest of experiments, if not otherwise stated.

Tolerance	Dimension 3(E)	3(MBR)	5(E)	5(MBR)	8(E)	8(MBR)
0.5	5.45e12	2.26e12	6.46e21	2.80e26	8.56e48	2.77e45
0.3	5.01e12	2.28e12	5.21e21	2.81e26	3.94e48	1.56e46
0.1	4.11e12	2.31e12	8.77e20	1.26e25	1.08e49	5.79e45
0.05	3.38e12	2.31e12	2.67e21	2.61e26	3.99e48	1.14e46
0.03	3.08e12	2.36e12	5.08e20	1.27e25	1.29e48	1.35e46
0.01	2.82e12	2.35e12	1.69e21	2.88e26	1.57e48	1.06e45
R-tree	1.69e17		2.03e28		1.60e44	

Table 6.3: Volume of the pre-leaf nodes.

Fanout	Dimension 3	4	5	8	10	12
Inner node	146	113	93	60	48	40
InneLeaf	39	26	19	9	6	4
Leaf node	255	204	170	113	92	78

Table 6.4: THe node fanout.

6.3.3 Node Fanout

In Table 6.4 we can study the fanout of the inner, pre-leaf and leaf nodes of the eR-tree. Node size was set to 4096KB. The R-tree's inner node fanout is same as fanout of inner nodes of eR-tree (both holds only MBR), the same holds also for leaf nodes. A poor fanout of the *pre-leaf* nodes is caused by huge space requirements of ellipsoids. To store ellipsoid we need to store $dimension \times dimension$ symmetric shape matrix and $dimension \times 1$ center of the ellipsoid. Taking in account, that the elements of the shape matrix are of double type (8 bytes) we will end up with $(((dimension + 1) * dimension * 8/2) + (dimension * 8) + (2 * dimension * 4))$ bytes required to store one pre-leaf node item. First addend is for shape matrix, second for ellipsoid center, third for MBR. Using float type instead of double should double the pre-leaf nodes fanout.

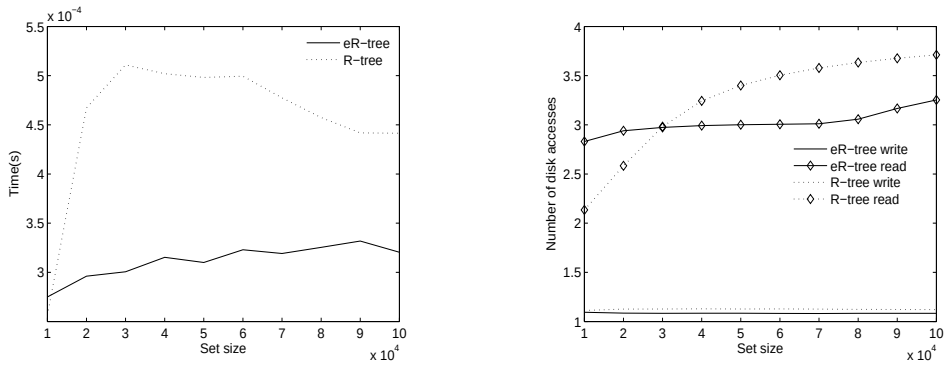
The node fanout directly affects the performance of the tree, as smaller fanouts lead to higher trees and obviously more I/O. In Table 6.5 are shown heights of indices from section 6.3.1. Columns with (*) denotes the theoretical height with 100% node utilization.

Data set	Height			
	eR-tree	R-tree	eR-tree(*)	R-tree(*)
SCU1x4	2	2	2	1
SCU4x4	2	3	2	2
SCU8x4	2	4	2	2
SCU1x5	2	3	2	2
SCU5x5	4	8	3	2
SCU1x6	4	7	3	2
SCU2x6	5	8	3	2
IMDB	3	3	3	2
TRAJ	2	2	2	2

Table 6.5: Tree height.

6.3.4 Impact of a data set size on indexing speed

In this section, we present the average cost of inserting a new entry into eR-tree index holding various amount of data. In Figure 6.5 are shown results measured on the SCU1x5 data set. In subfigure (a) is plotted average CPU time required for insertion of one new entry. Results for I/O are given in subfigure (b). Read and writes are measured separately. We can observe, that eR-tree requires $\approx 30\%$ less CPU compared with R-tree when indexing SCU1x5.

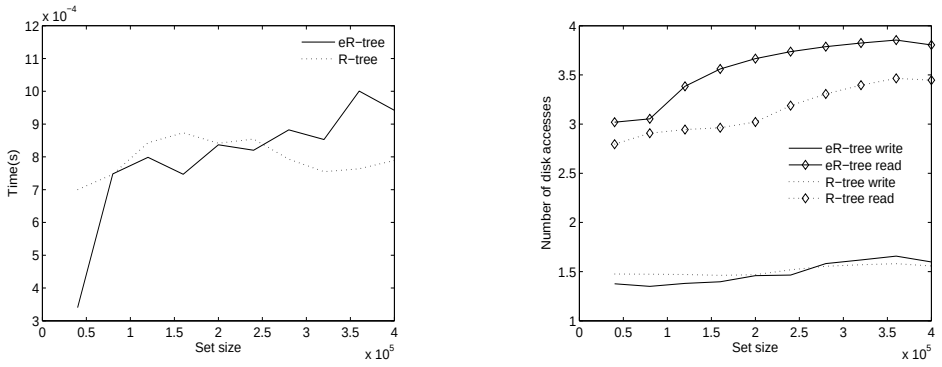


(a) CPU time (b) Number of disk accesses

Figure 6.5: Insertion cost on SCU1x5 data set with dimension equal to 5.

In Figure 6.6 is presented an average cost of insertion of a new entry of IMDB data set. I/O costs are approximately the same for the eR-tree and

the R-tree. Noticeable is a fact, that the absolute CPU time is $\approx 50\%$ higher when indexing IMDB data set compared with SCU1x5 (both for eR-tree and R-tree; insert of one new entry of the SCU1x5 data set takes $\approx 4x10^{-4}$ and insert of one new entry of the IMDB data set takes $\approx 8x10^{-4}$). I/O keeps about the same level when comparing IMDB and SCU1x5 data sets.



(a) CPU time (b) Number of disk accesses

Figure 6.6: Insertion cost on IMDB data set with dimension equal to 5.

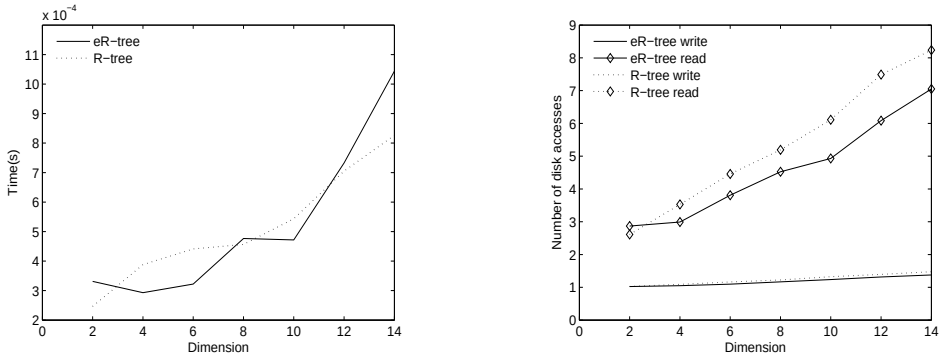
6.3.5 Impact of a dimension on indexing speed

Measurements of the insertion cost of a new entry as a parameter of the dimension could by found in Figure 6.7 and 6.8. In the first figure, results of SCU1x5 data set are presented. Measured dimensions were 2,4,6,8,10,12,14. In subfigure (a) average CPU time required to insert one new entry is depicted. In subfigure (b) average reads and writes are show. CPU usage is approximately equivalent for eR-tree and R-tree. In terms of I/O, eR-tree slightly overcomes R-tree.

In the second Figure 6.8 results for IMDB data set are presented. Dimensions 1, 2, 3, 4 were tested. We can observe that in 1 dimensional case R-tree is significantly faster than eR-tree.

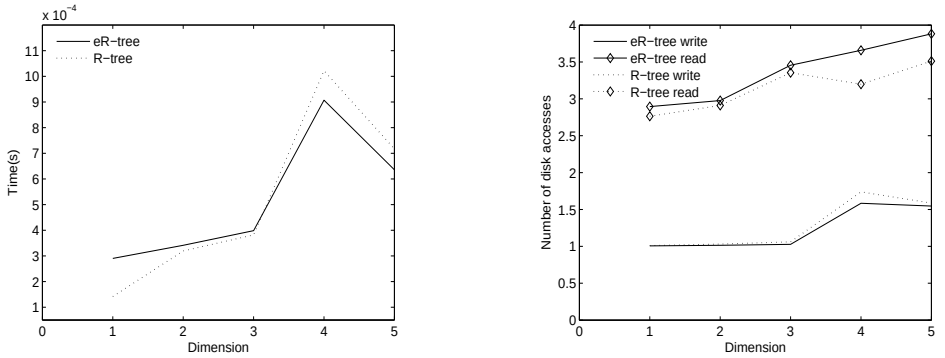
6.3.6 Other interesting facts

For the sake of completeness, in Table 6.6 are provided counts of ellipsoid constructions when indexing different data sets. In parenthesis is specified dimensionality of indexed set. We can observe, that dimensionality has high



(a) CPU time (b) Number of disk accesses

Figure 6.7: Insertion cost on SCU1x5 data set with varying dimension.



(a) CPU time (b) Number of disk accesses

Figure 6.8: Insertion cost on IMDB data set with dimension 1, 2, 3, 4.

	Data Set(dimension)				
#	SCU1x4(5) 272	SCU4x4(5) 1328	SCU8x4(5) 2874	SCU1x5(5) 3041	SCU5x5(5) 19867
#	SCU1x6(5) **36682**	SCU2x6(5) 78503	IMDB(5) 33710	TRAJ(5) 16607	
#	SCU1x5(2) 1023	SCU1x5(6) 5153	SCU1x5(10) 15638	SCU1x5(12) 27986	SCU1x5(14) **36748**

Table 6.6: Number of ellipsoid constructions when indexing.

impact on the number of ellipsoid constructions. E.g. approximately same number of ellipsoid constructions is required when indexing $1x10^6$(5 dim) and $1x10^5$(14 dim) data sets.

6.4 Querying

6.4.1 Query set

To analyze a query performance different query sets were generated in advance. One query set consist of 1000 individual queries. Results for one query set are presented as the average of all individual queries included in the set. Each query set is characterized by a *selectivity*. The selectivity of the query denotes the number of hits it returns. All individual queries included in query set have equal selectivity. It is important to have selectivity fixed, when e.g. studying I/O costs of the index with varying size, as the I/O costs are closely related to the number of hits returned by the query.

We will explore the eR-tree properties with query sets with selectivity equal to 3, if not otherwise stated.

6.4.2 Impact of a data set size on a querying speed

In this section, we will study the impact of the data set size on querying speed. Data sets SCU1x4 to SCU2x6 were indexed and then queried. Results are presented in Figure 6.9. In the subfigure (a) and (c) is presented execution time – average CPU time required to evaluate one query of the query set of selectivity 3 and 50. The eR-tree appears to be outperformed by the R-tree. R-tree in average requires only 75% of the time required by eR-tree. In the

left subfigure (b) and (d) are drawn I/O costs. On the left y-axis is shown total number of the I/O required to fulfill the query. On the right y-axis is shown the number of leaf nodes searched during the query evaluation. Here we can notice in contrary to CPU costs that the eR-tree outperforms R-tree. The eR-tree requires in average only 72% of R-tree I/O operations.

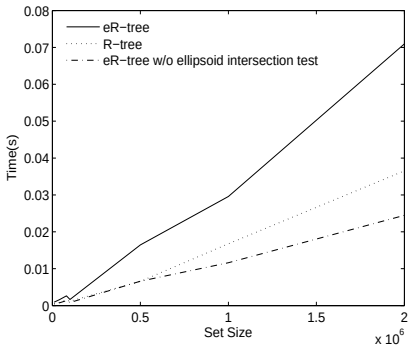
Now comes into question: Why is eR-tree slower in terms of CPU time, but faster in terms of the I/O? We believe that the answer is that the eR-tree is slower in terms of CPU time because the cost of the test for ellipsoid and query window intersection is much more expensive, than the cost of the test for MBR and query window intersection. This inefficiency was expected and is the reason why ellipsoids are only in the pre-leaf nodes, not in all inner nodes. In addition, we have to admit that the poor I/O results of the R-tree are partially results of the worse space utilization compared with the eR-tree (revisit section 6.3.1, R-tree requires more nodes to accommodate the same data). To prove our believe, we have disabled the ellipsoid intersection test (only MBR of the pre-leaf nodes are tested for intersection with query window) and rerun the experiment again. The CPU time dropped significantly, as can be observed in subfigure 6.9 (a).

Why eR-tree requires *significantly* less I/O operations? Differences in space utilization can be eliminated because the differences are negligible and cannot lead to so significant difference in querying I/O costs. Other explanation can be that many regions are not searched (saved I/O) because of ellipsoids. I.e. query window intersect with MBR in the pre-leaf node, but not with ellipsoid. In Table 6.7 are provided average counts of pre-leaf nodes encountered during query evaluation, where MBR do intersect query window, but ellipsoid don't (revisit motivation in Figure 3.1). The counts are also insignificant compared to total I/O ($12.6 \ll 375$) and cannot be the reason of the differences in querying I/O costs. We believe that the true explanation is in algorithm of leaf node splitting.

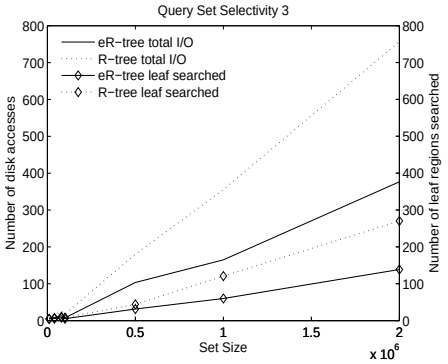
	Query Set						
	SCU1x4	SCU4x4	SCU8x4	SCU1x5	SCU5x5	SCU1x6	SCU2x6
#	0.03	0.06	0.1	0.02	3.2	5.5	12.6

Table 6.7: Number of pre-leaf nodes not searched due to ellipsoids for SCU data set.

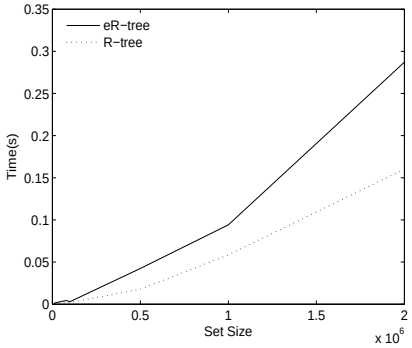
Figure 6.10 depicts query costs for IMDB data set. Subset with $4x10^4$, $8x10^4$, $12x10^4$, $16x10^4$, $20x10^4$, $24x10^4$, $28x10^4$, $32x10^4$, $36x10^4$, $40x10^4$ entries were indexed. Average CPU time required to evaluate one query of the query set



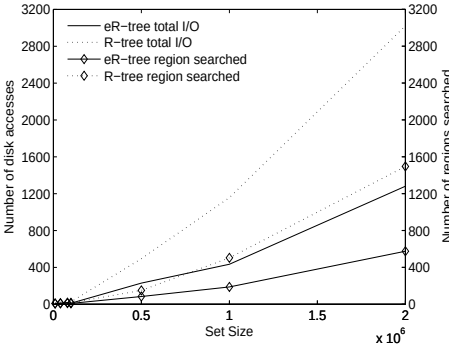
(a) CPU time, Selectivity=3



(b) Number of disk accesses, Selectivity=3



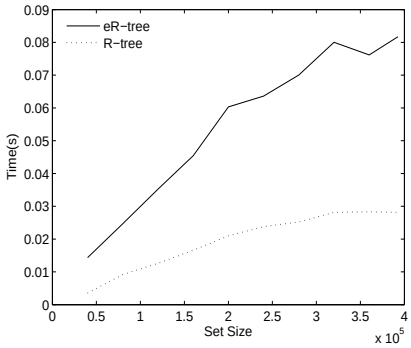
(c) CPU time, Selectivity=50



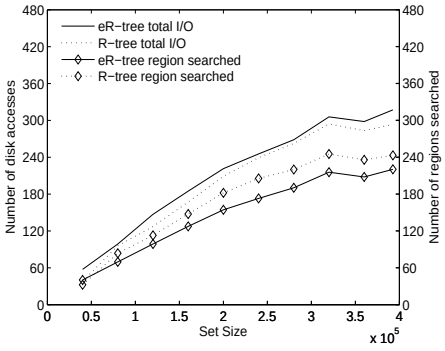
(d) Number of disk accesses, Selectivity=50

Figure 6.9: Querying cost as a parameter of set size. Results for SCU data set with dimension equal to 5.

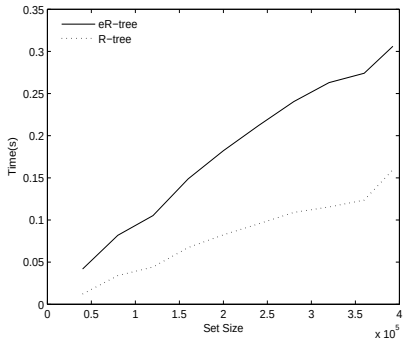
of selectivity 3 and 50 is shown in the subfigure (a) and (c). The eR-tree is again outperformed by the R-tree (R-tree requires in average only 65% of the eR-tree time). However in the subfigure (d) we can see, that eR-tree outperforms R-tree in terms of I/O. E.g. on the whole IMDB data set eR-tree requires only 1049 I/O operations compared to R-tree's 1410 operation to evaluate the same query. As the data set size increases, the discrepancy between eR-tree and R-tree I/O grows.



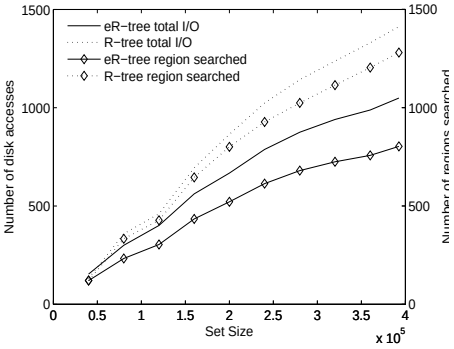
(a) CPU time, Selectivity=3



(b) Number of disk accesses, Selectivity=3



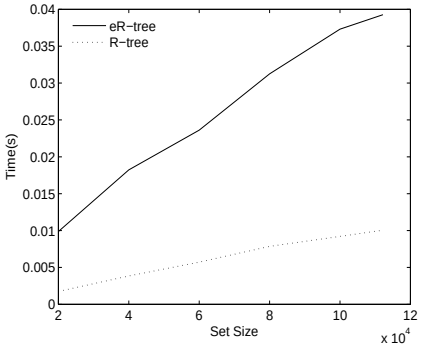
(c) CPU time, Selectivity=50



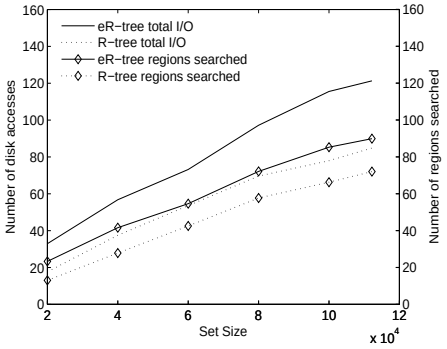
(d) Number of disk accesses, Selectivity=50

Figure 6.10: Querying cost as a parameter of set size. Results for IMDB data set with dimension equal to 5.

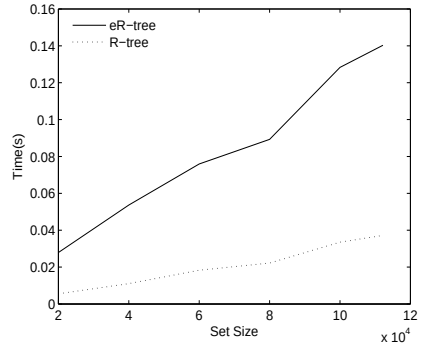
For the sake of completeness in Figure 6.11 are shown querying costs for the trajectory data set.



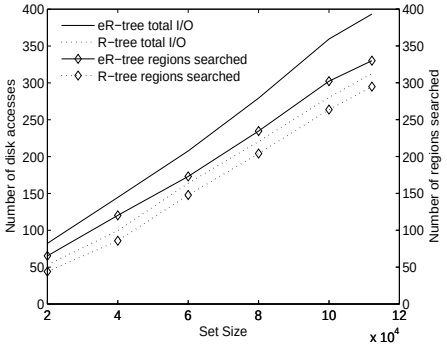
(a) CPU time, Selectivity=3



(b) Number of disk accesses, Selectivity=3



(c) Number of disk accesses, Selectivity=3



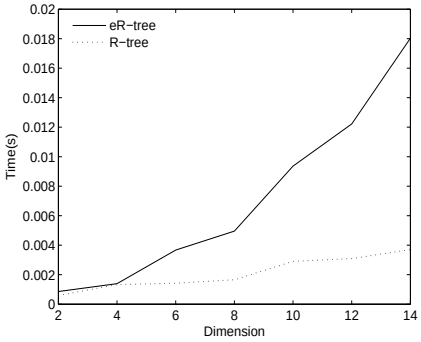
(d) CPU time, Selectivity=50

Figure 6.11: Querying cost as a parameter of set size. Results for the trajectory data set with dimension equal to 4.

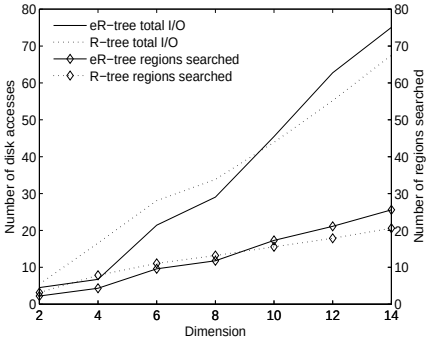
6.4.3 Impact of a dimension on a querying speed

How does dimension affect the query performance? To answer this question SCU1x5 data set in dimension 2, 4, 6, 8, 10, 12, 14 was indexed and than queried. The results presented in Figure 6.12 clearly shows that in higher dimensions eR-tree cannot by compared with R-tree as it require more time and more I/O operations. In the left subfigure (a) and (c) we can see that CPU time grows exponentially. The explanation is simple. As the dimension increases, the test of ellipsoid and query window intersection becomes more and more expensive. Recall, that this test is solved as an optimization problem and in higher dimension there are "more" directions in which this optimization can "go". In the right subfigure (b) and (d) we can notice, that

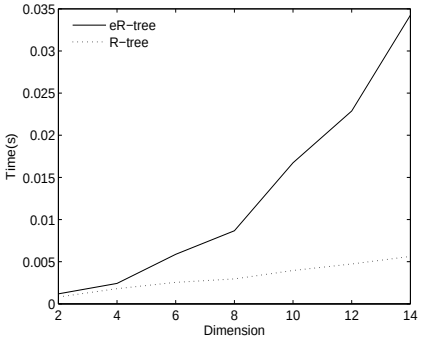
the eR-tree requires also more I/O. This is also the result of mentioned intersection test. The optimization can execute at most 60 iterations. If it does not manage to solve the problem in those 60 iterations, the optimization is stopped with return value `true`(i.e. ellipsoid do intersect query window). It is cheaper to search a region, than leave optimization to go through e.g. 500 iterations. Just for comparison, full scan with dimension equal to 14 would require 1470 I/O operations compared with 120 I/O required by eR-tree – i.e. it is still worth to use index.



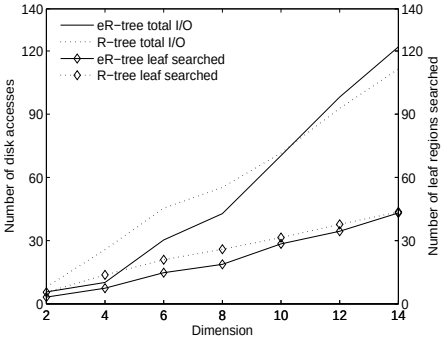
(a) CPU time, Selectivity=3



(b) Number of disk accesses, Selectivity=3



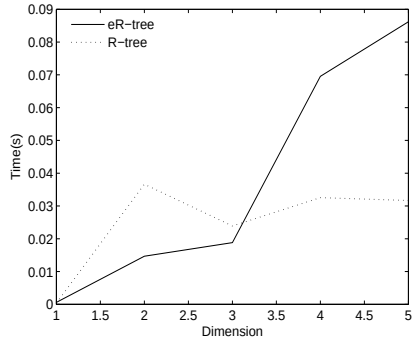
(c) CPU time, Selectivity=50



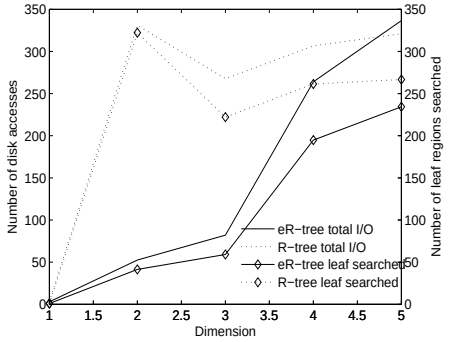
(d) Number of disk accesses, Selectivity=50

Figure 6.12: Querying cost as a parameter of dimension. SCU1x5 data set with dim=2, 4, 6, 8, 10, 12, 14.

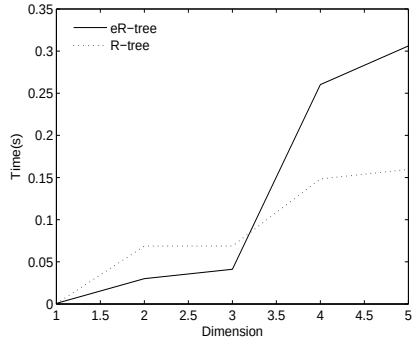
In Figure 6.13 are shown measured values for IMDB data set with dimension 1, 2, 3, 4, 5. We can conclude, that eR-tree is more suitable for data with low dimensionality.



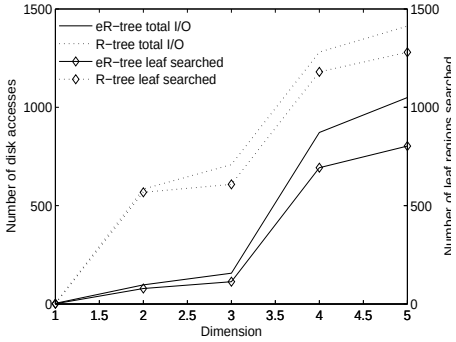
(a) CPU time, Selectivity=3



(b) Number of disk accesses, Selectivity=3



(c) CPU time, Selectivity=50

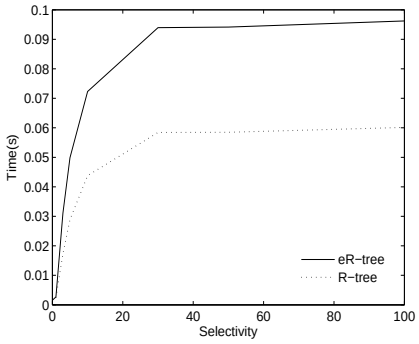


(d) Number of disk accesses, Selectivity=50

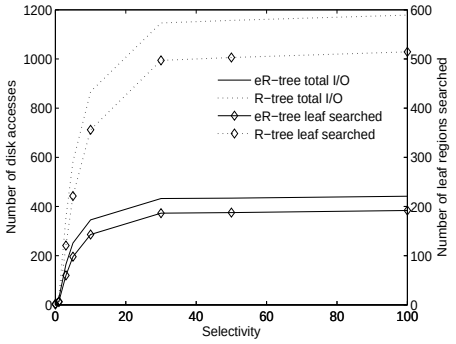
Figure 6.13: Querying cost as a parameter of dimension. IMDB data set with dim=1,2,3,4,5.

6.4.4 Impact of a selectivity on a querying speed

How does selectivity affect query performance? In previous experiments, we could see differences between query set of selectivity 3 and selectivity 50. To thoroughly investigate impact of selectivity on query performance, SCU1x6 data set was queried with query sets of selectivity 0, 1, 3, 5, 10, 30, 50, 100. The results are presented in Figure 6.14. It can be seen, that eR-tree is taking advantage of higher selectivity. The same can be observed in Figure 6.15 where IMDB data set is subject of evaluation.

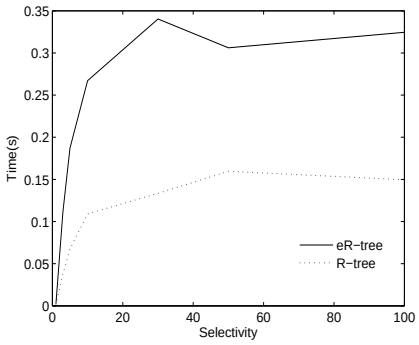


(a) CPU time

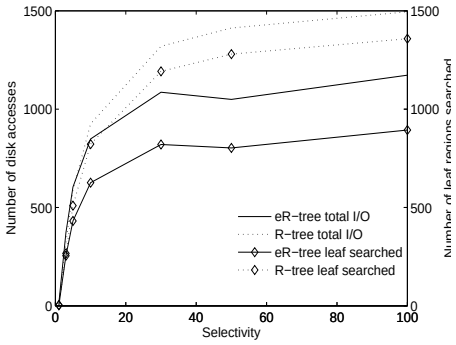


(b) Number of disk accesses

Figure 6.14: Querying cost as a parameter of a selectivity. SCU1x6 data set used.



(a) CPU time



(b) Number of disk accesses

Figure 6.15: Querying cost as a parameter of a selectivity. IMDB data set used.

Chapter 7

Conclusion

The goal of this thesis was to investigate the applicability of ellipsoids to the R-tree index structure.
In section 4.2.2 and 6.3 we have proven with our experimental results that a minimum volume covering ellipsoid construction is efficient enough to be incorporated into R-tree index structure and therefore the idea of R-tree with ellipsoidal regions is well worth of considering. Taking into account attributes of the ellipsoids and having results of a deep survey of various R-tree variants (Chapter 2), we have proposed our eR-tree (**e**llipsoid R-tree) in Chapter 3, which was later the subject of thorough experiments. The aim of all experiments was to confirm the benefits of ellipsoids in R-tree, reveal their possible demerits and eventually to point out the direction of further investigation.
We have found out, that eR-tree significantly outperforms R-tree in terms of I/O on sparse clustered data, where ellipsoidal regions are superior to minimum bounding rectangles, because they tend to cover less dead space and produce less overlaps. On data of high density, this advantage of ellipsoids is being suppressed.
The most significant demerit of ellipsoids is a very expensive test of intersection with query window (section 5.2.2). The complexity of this test mostly overcomes savings of the I/O operations. The second problem of ellipsoids is their vast storage requirements (section 6.3.3), which degrade fanout of the nodes leading into higher trees and bigger indices.
Considering demerits of ellipsoids, we still believe R-tree with ellipsoidal regions is worth of further investigation. Attention should be paid to optimization of ellipsoid and query window intersection test and splitting algorithms. All further investigations should be in favor of a sparse data, where ellipsoids overcome minimum bounding rectangles.

List of Algorithms

List of Tables

List of Figures

Bibliography

[Bar82] E. Barnes. An algorithm for separating patterns by ellipsoids. *Image Processing and Pattern Recognition*, 26(6):759, 1982.

[BBK01] Christian Böhm, Stefan Berchtold, and Daniel Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):321–373, September 2001.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 322–331. ACM Press, 1990.

[Boy04] Stephen P. Boyd. *Convex Optimization*. Cambridge University Press, 2004. `http://www.stanford.edu/~boyd/cvxbook/`.

[Gro03] Amphora Research Group. *Amphora Tree Object Model (ATOM) Book*, 2003.

[GS97] Bernd Gärtner and Sven Schönherr. Smallest enclosing ellipses - fast and exact. *Report B 97-03*, 1997.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In Beatrice Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57. ACM Press, 1984.

[KS97] Norio Katayama and Shin'ichi Satoh. The sr-tree: An index structure for high-dimensional nearest neighbor queries. In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 369–380. ACM Press, 1997.

[KT93] Leonid G. Khachiyan and Michael J. Todd. On the complexity of approximating the maximal inscribed ellipsoid for a polytope. *Mathematical Programming: Series A and B*, 61:137 – 159, September 1993.

[KY05] Piyush Kumar and E. Alper Yildirim. Approximate minimum volume enclosing ellipsoids using core sets. *Journal of Optimization Theory and Applications*, 1:1–21, July 2005.

[Mos05] Nima Moshtagh. Minimum volume enclosing ellipsoid. *Convex Optimization*, 2005.

[Mur01] Katta G. Murty. *Computational and Algorithmic Linear Algebra and n-Dimensional Geometry*, chapter Quadratic Forms, Positive, Negative (Semi) Definiteness. Online WebBook, 2001.

[SM04] Peng Sun and Robert M. Freund. Computation of minimum volume covering ellipsoids. *Discrete Applied Mathematics*, 52:690706, September October 2004.

[SRF87] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *VLDB*, pages 507–518, 1987.

[TY07] Michael J. Todd and E. Alper Yildirim. On khachiyan's algorithm for the computation of minimum-volume enclosing ellipsoids. *Discrete Applied Mathematics*, 155:1731–1744, August 2007.

[Van06] Robert J. Vanderbei. Loqo users manual version 4.05. Technical Report ORFE-99, Operations Research and Financial Engineering, Princeton University, 2006.

[Wel91] Emo Welzl. Smallest enclosing disks(balls and ellipsoids). *Lecture Notes in Computer Science*, pages 359 – 370, 1991.

[WJ96] David A. White and Ramesh Jain. Similarity indexing with the ss-tree. In *ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering*, pages 516–523, Washington, DC, USA, 1996. IEEE Computer Society.

[Yil06] E. Alper Yildirim. On the minimum volume covering ellipsoid of ellipsoids. *SIAM J. on Optimization*, 17(3):621–641, 2006.

Appendix A

Visualization

How much overlaps does the eR-tree produce? How much dead space it covers? How does partitioning of the indexed space really look like? How it differs from the R-tree? All this questions are answered in the most straightforward way – by an image. Selected data sets (or its portions) are plotted, each in separate section, along with their eR-tree and R-tree indices.

A.1 SCU4x4 data set

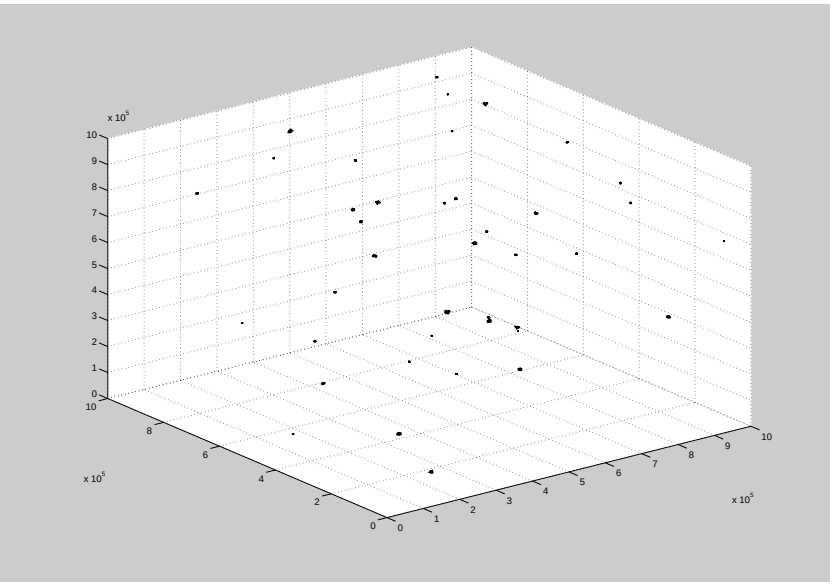


Figure A.1: Plot of SCU4x4 data set holding 40000 entries.

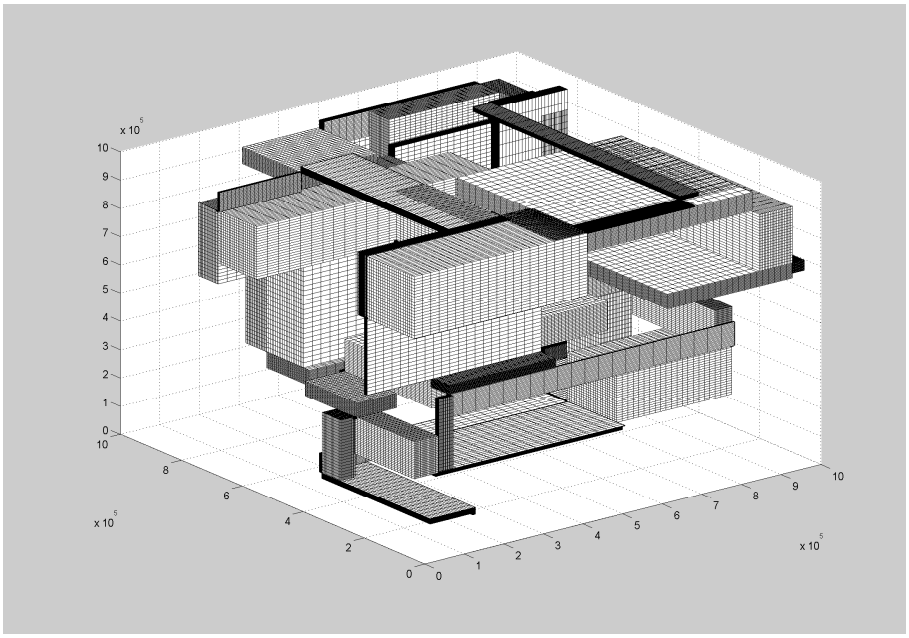


Figure A.2: MBRs of R-tree's pre-leaf nodes holding SCU4x4 data set.

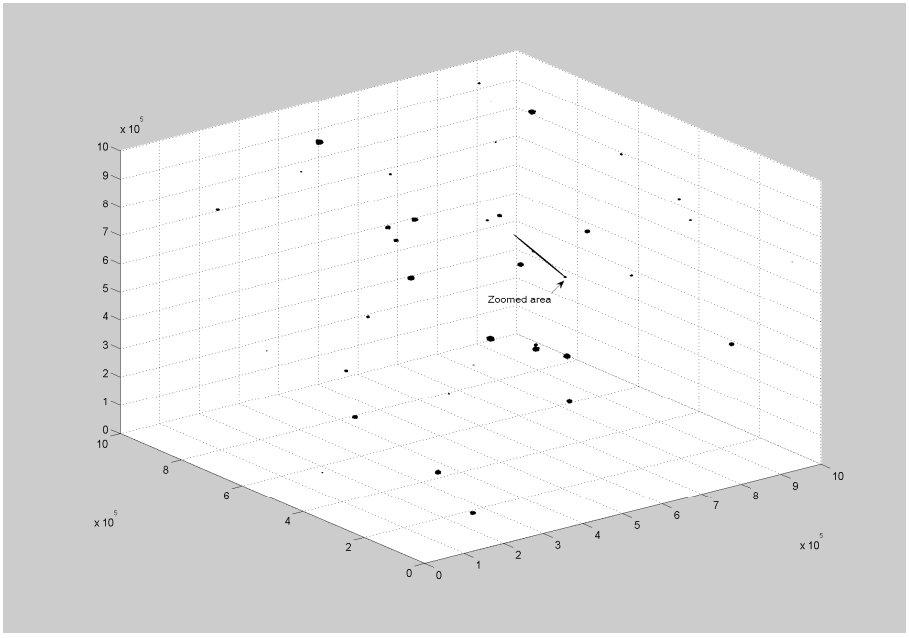


Figure A.3: Ellipsoids of eR-tree's pre-leaf nodes holding SCU4x4 data set.

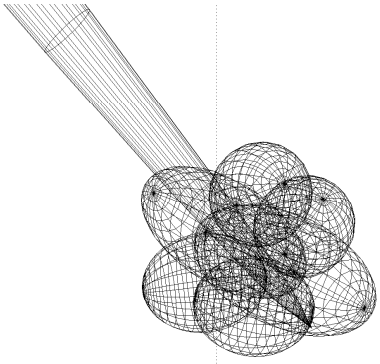


Figure A.4: Zoomed area of Figure A.3.

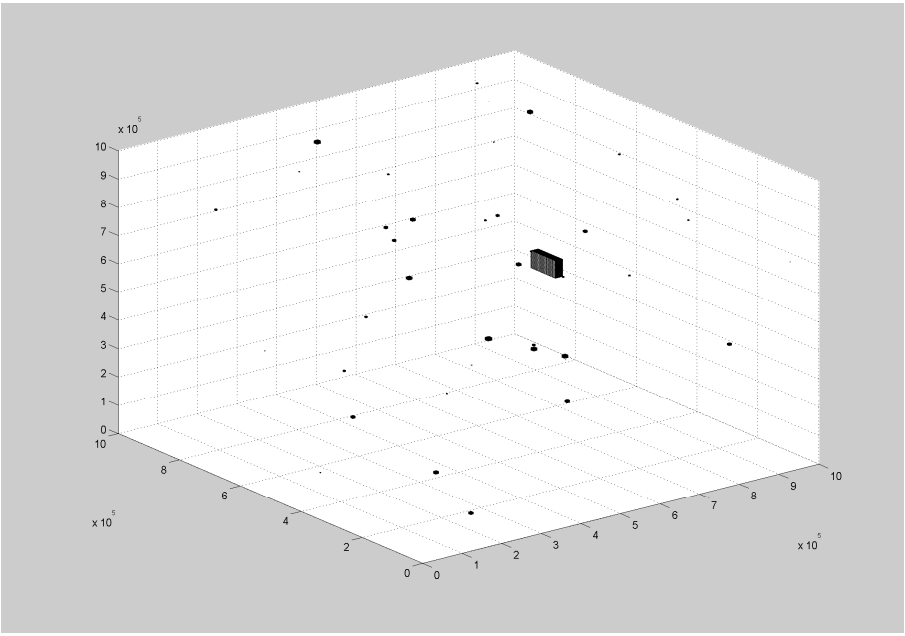


Figure A.5: MBRs of eR-tree's pre-leaf nodes holding SCU4x4 data set.

A.2 IMDB data set

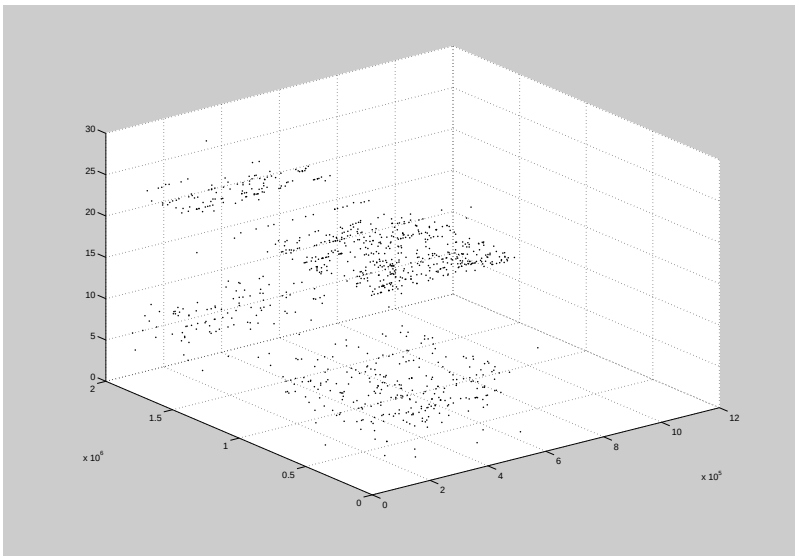


Figure A.6: Plot of first 20000 entries of IMDB data set.

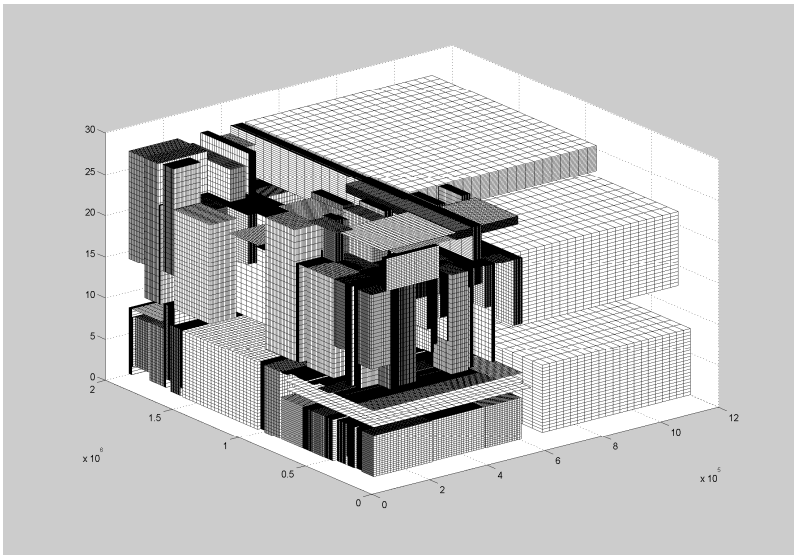


Figure A.7: MBRs of R-tree's pre-leaf nodes holding first 20000 entries of IMDB data set.

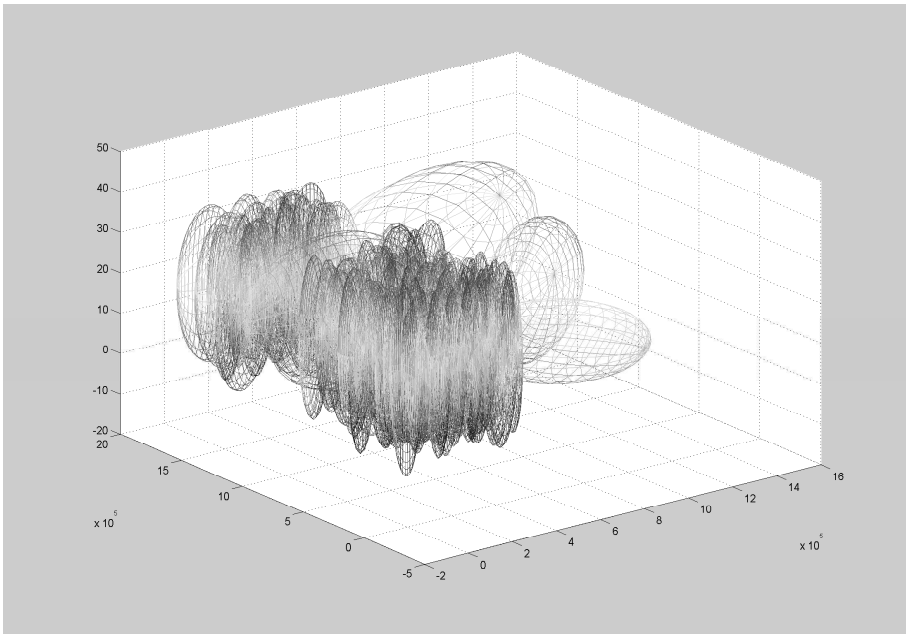


Figure A.8: Ellipsoids of eR-tree's pre-leaf nodes holding first 20000 entries of IMDB data set.

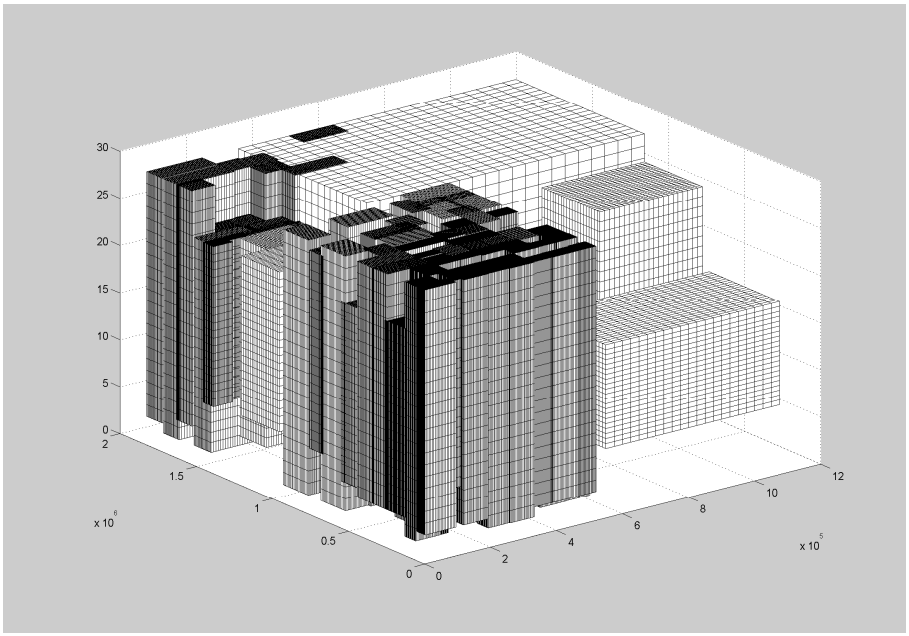


Figure A.9: MBRs of eR-tree's pre-leaf nodes holding first 20000 entries of IMDB data set.

A.3 Trajectory data set

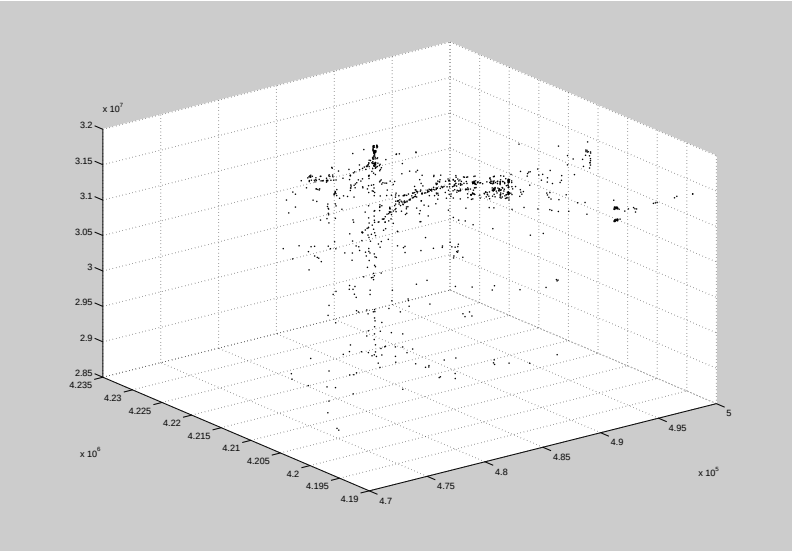


Figure A.10: Plot of first 20000 entries of Trajectory data set.

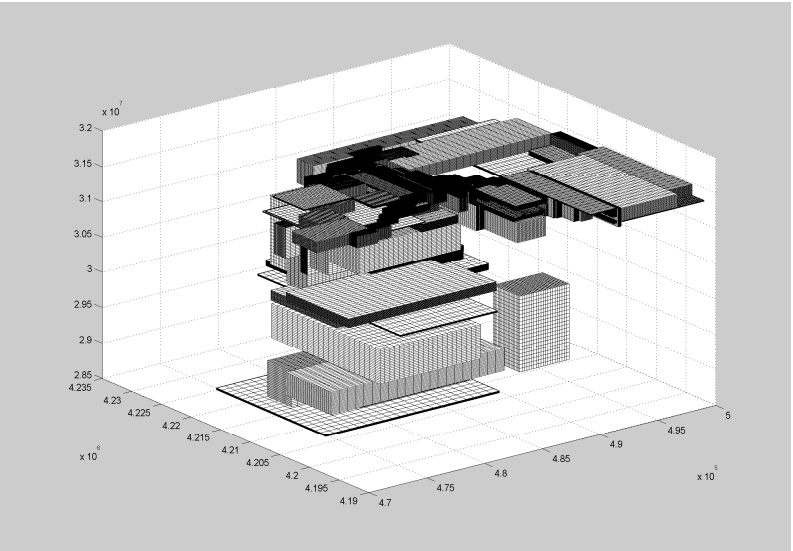


Figure A.11: MBRs of R-tree's pre-leaf nodes holding first 20000 entries of Trajectory data set.

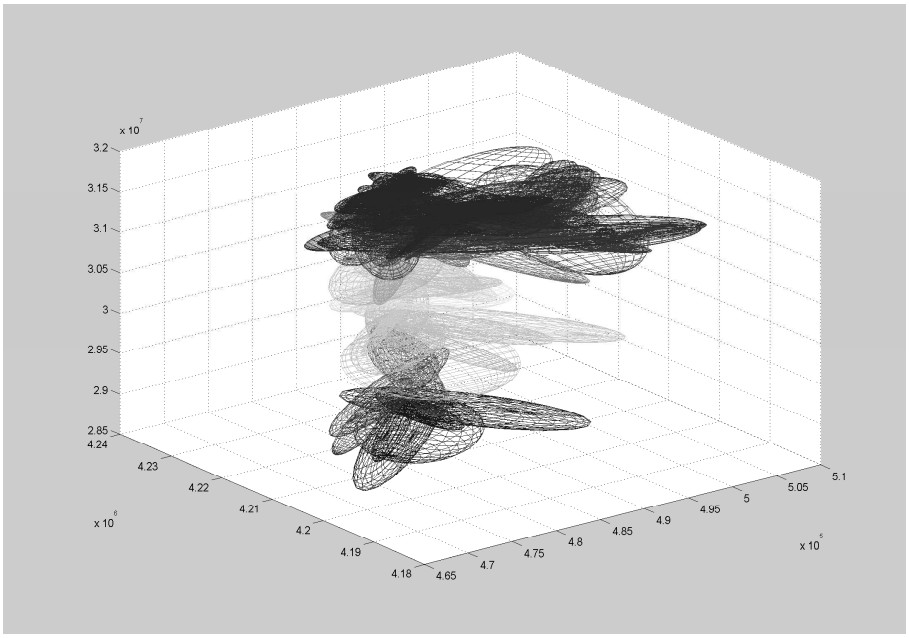


Figure A.12: Ellipsoids of eR-tree's pre-leaf nodes holding first 20000 entries of Trajectory data set.

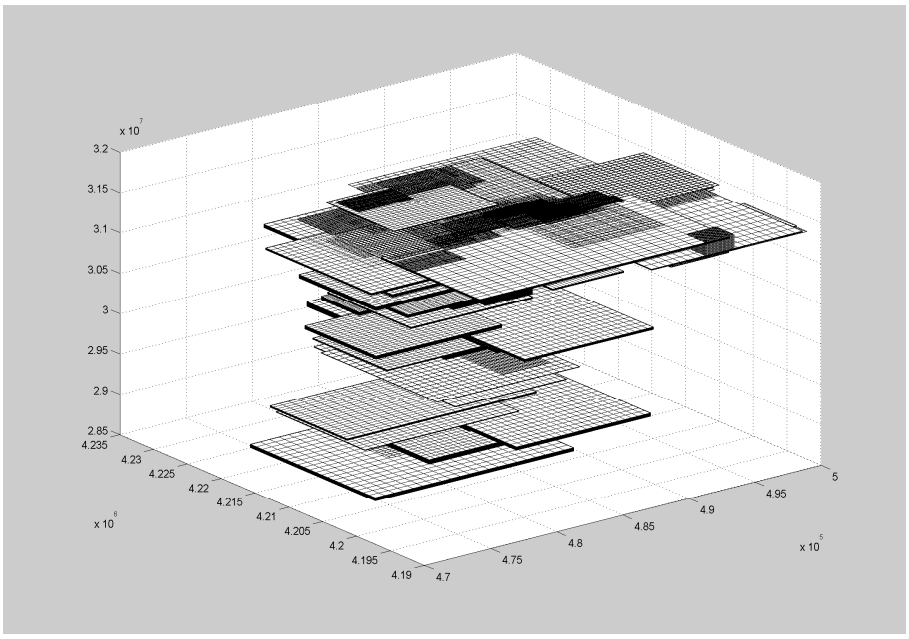


Figure A.13: MBRs of eR-tree's pre-leaf nodes holding first 20000 entries of Trajectory data set.

Appendix B

Organization of attached compact disk

B.1 Source Code

Source code is provided as Visual C++ 2005 project. eR-tree source code can be found in /Source/eRtree directory. R-tree source code can be found in /Source/Rtree directory.

B.2 Data Sets

Data sets are provided in /DataSets directory in "avs" file format.

B.3 Query Sets

All query sets can by found in /QuerySets directory. Query set name is of form qSet[DATASETNAME][DATASETSIZE][SELECTIVITY] where DATASETNAME specifies for which data set was query set generated. DATASETSIZE denote size of data set size (if only portion of data set was indexed, i.e. when impact of data set size on querying speed was examined). SELECTIVITY denotes selectivity of the query set.

B.4 Experimental Results

Log files of all experiments are included and contain some additional information, which might be of interest. Log files for eR-tree are called eR.log. Log files for R-tree are called R.log. In the first line of log file is written either

name of data set being indexed, or name of query set that was issued against the index.

Indexing log file

Index of each 100^{th} element is logged into indexing log file. In case of eR-tree indexing log file, ellipsoid constructions (e.g. Approximating 278(279) tuples by khachiyanMVCE) are logged together with leaf node splitting events (Split dimension 0 with splitOrder 270 choosen). At the end of the log file is written time required to index data set, followed by volume of regions of pre-leaf nodes, tree statistics and cache statistics. Statistics are self-explanatory.

Querying log file

First line of querying log file is followed by tree statistics. Next lines are of form x (.*), where x is the number of hits returned by one query of query set. .* specifies lower corner of query window. In the end of the log files are written summary (for whole query set) cache and query statistics, which should be also self-explanatory. Querying log file names can be suffixed by e.g. S050, meaning that the query set of selectivity equal to 50 was used. If the log name is not suffixed with selectivity specification, implicit selectivity of 3 is assumed.

Organization

/ExperimentalResults/StoppingCriterion directory contains results of experiments presented in section 6.3.2. Stopping criterion is a suffix of a log file, e.g. 0_05 denote stopping criterion of value 0.05.

/ExperimentalResults/Fanout directory contains results of experiments presented in section 6.3.3.

/ExperimentalResults/SetSizeVsIndexingSpeed directory contains results of experiments presented in section 6.3.4. Subdirectory names denote set size.

/ExperimentalResults/DimensionVsIndexingSpeed directory contains results of experiments presented in section 6.3.5. Subdirectory names denote dimension.

/ExperimentalResults/SetSizeVsQueryingSpeed directory contains results of experiments presented in section 6.4.2. Subdirectory names denote set size.

/ExperimentalResults/DimensionVsQueryingSpeed directory contains results of experiments presented in section 6.4.3. Subdirectory names denote dimension.

/ExperimentalResults/Selectivity directory contains results of experiments presented in section 6.4.4. Subdirectory names denote selectivity of query set.