

# Elliptic Indexing of Multidimensional Databases

Ondrej Danko<sup>1</sup>

Tomáš Skopal<sup>2</sup>

<sup>1</sup> Comenius University in Bratislava, FMUK, Department of Information Systems  
Odbojárov 10, 820 05 Bratislava, Slovak Republic  
Email: [ondrej.danko@fm.uniba.sk](mailto:ondrej.danko@fm.uniba.sk)

<sup>2</sup> Charles University in Prague, FMP, Department of Software Engineering  
Malostranské nám. 25, 118 00 Prague, Czech Republic  
Email: [skopal@ksi.mff.cuni.cz](mailto:skopal@ksi.mff.cuni.cz)

## Abstract

In this work an R-tree variant, which uses minimum volume covering ellipsoids instead of usual minimum bounding rectangles, is presented. The most significant aspects, which determine R-tree index structure performance, is an amount of dead space coverage and overlaps among the covering regions. Intuitively, ellipsoid as a quadratic surface should cover data more tightly, leading to less dead space coverage and less overlaps. Based on studies of many available R-tree variants (especially SR-tree), the eR-tree (ellipsoid R-tree) with ellipsoidal regions is proposed. The focus is put on the algorithm of ellipsoids construction as it significantly affects indexing speed and querying performance. At the end, the eR-tree undergoes experiments with both synthetic and real datasets. It proves its superiority especially on clustered sparse datasets.

## 1 Introduction

In the last decades, the demand for efficient searching in large multimedia databases has begun emerging much more often than anytime before. Especially the applications from areas like medicine, geography or CAD experienced an absence of techniques which would enable them to efficiently search in protein databases, geographical maps or CAD datasets. Because the nature of geographical or medicine data differs, a custom solution would be always required to provide efficient retrieval. To solve this problem, simple, yet powerful idea is applied – *feature transformation*. Each object of a dataset is transformed into a tuple of  $n$ -dimensional space representing that object (based on certain transformation rules). Afterwards, a multidimensional indexing technique is employed to enable fast retrieval. The most obvious query type in multidimensional indexing is *window query* (or range query). A window query simply specifies some portion of the  $n$ -dimensional space, a window (set of intervals on all dimensions), and returns all dataset tuples inside.

## 2 Related Work

To efficiently search for tuples inside a query window, we have to employ a *spatial access method* (SAM, or called a multidimensional index). In this section

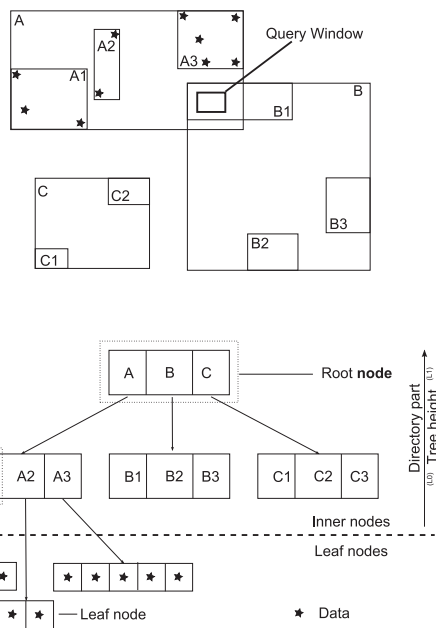


Figure 1: R-tree index structure.

we give an overview of the most successful SAMs, all based on R-tree.

### 2.1 R-tree

The R-tree (Guttman 1984) is a height-balanced multidimensional index structure similar to B-tree. The basic idea behind the R-tree is to hierarchically partition the search space into nested regions. The space partitioning is neither complete nor disjoint, while the nested hierarchies are formed as paths from the root node to the leaf nodes. The root node region encloses regions of all its child nodes. The R-tree consists of inner and leaf nodes. A leaf node holds the data tuples and consists of entries  $\langle I, data \rangle$ , where  $I$  represents a minimum bounding rectangle (MBR) of *data*. Similarly, an inner node consists of entries  $\langle I, child\_pointer \rangle$  where  $I$  is MBR of all entries contained in node referenced by *child\_pointer*, see Figure 1. Searching in R-tree index structure means traversing those paths of the tree that intersect with the search region (query window in our case).

### 2.2 Other SAM

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

- minimization of leaf level node region overlaps

- minimization of leaf level regions surface
- minimization of volume of inner nodes
- maximization of storage utilization

### 2.2.1 R\*-tree

The R\*-tree (Beckmann et al. 1990) introduces *forced reinsertions*, because R-tree-based structures are highly sensitive to the order in which are tuples inserted. When a leaf node becomes overfull, some portion of its member tuples which are most distant from the center of the node's MBR are deleted from index and reinserted again. This improvement pushed the storage utilization to 71% -76%. The R\*-tree also prefers squared MBRs over rectangular.

### 2.2.2 R<sup>+</sup>-tree

The key idea behind the R<sup>+</sup>-tree (Sellis et al. 1987) is an overlap-free splitting in the tree directory (the inner nodes). Generally, there is no guarantee that such splitting exists. In case there exists no overlap-free splitting, the R<sup>+</sup>-tree introduces a *forced splitting*. When considering the example in Figure 1, to get rid of overlap between the A and B regions, we necessarily need to split the region B1. 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. As a side effect of forced splits, unlike regular R-tree, the R<sup>+</sup>-tree also cannot guarantee 50% space utilization. On the other hand, the authors of R<sup>+</sup>-tree claim their modification requires 50% less page accesses (on average) to process a query, compared to the R-tree.

### 2.2.3 SS-tree

The SS-tree (White & Jain. 1996) was introduced to support similarity searches (meaning nearest neighbor queries) in higher dimensions (thus **S**imilarity **S**earch-**t**ree). The SS-tree uses spheres instead of MBRs as page regions. When comparing properties of MBRs with spheres, it should be said that:

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

When using spheres, the first mentioned property decreases the performance of window queries, while the second one favors the nearest neighbor queries. Another advantage of spheres is that they require less space to be stored than MBRs. For MBR we need to store two  $n$ -dimensional vectors, while for sphere it is enough to store one  $n$ -dimensional center and a one-dimensional diameter. This allows higher fanout of nodes, thus it eventually decreases the tree height. For performance reasons the spheres of SS-tree are not minimum volume spheres, but use centroids as their centers. Thus, the center of the sphere is computed as the average in each dimension of the data tuples being bounded. The diameter is then calculated so that it covers all tuples.

The SS-tree uses forced reinserting when an overflow is encountered; 30% of tuples with highest distance from the center of sphere are reinserted. While the storage utilization of R\*-tree is just 70-75%, the SS-tree reaches 85% on average. The splitting is based on variance. First, the dimension with highest variance is chosen. Then a split plane orthogonal to that dimension is found, so that the sum of variances in both, the new node and the old node,

is minimized. The authors of SS-tree claim the insertion uses significantly less CPU time, compared to R\*-tree (5-10x less). This is mainly because of simplistic insertion and linear split algorithm, compared to quadratic split algorithm of R\*-tree. When querying, the SS-tree outperforms the R\*-tree by a factor of two (approximately).

### 2.2.4 SR-tree

The SR-tree (Katayama & Satoh 1997) is merely a combination of SS-tree and R\*-tree. The authors presented an extensive comparison of MBRs and spheres properties. They defined a region in SR-tree as an intersection of MBR and sphere, gaining both advantages – a small volume of the MBR and a small diameter of the sphere. This extension should bring reasonable query performance for both window and nearest neighbor queries. The insertion and split algorithms are taken from the SS-tree and they are controlled solely by spheres. The SR-tree slightly outperforms both SS-tree and R\*-tree. As the most significant inefficiency of this approach, the authors discuss storage requirements of SR-tree region, which are 1.5× larger than that of R\*-tree and 3× larger than that of SS-tree.

### 2.2.5 X-tree

The X-tree (Berchtold et al. 1996) provides overlap-free split whenever it is possible, that is, just splits that do not lead to degeneration of the tree are allowed. Otherwise, the X-tree creates variable-sized directory nodes, so-called supernodes, to keep the hierarchy spatially compact. Since the supernodes can be quite large (which leads to partial sequential search), the X-tree could be seen as a hybrid of a linear array-like and a hierarchical R-tree-like directory. The main advantage is X-tree's performance when querying high-dimensional data, where it outperforms the R\*-tree by up to two orders of magnitude.

## 3 The eR-tree

As many studies (Beckmann et al. 1990, Katayama & Satoh 1997) indicate, the performance of the R-tree-based structures is mostly determined by the amount of region overlaps and dead space coverage. Most of the R-tree variants try to handle this problem by revisiting the splitting algorithms (e.g., the overlap-free one (Sellis et al. 1987)) or introducing concepts like forced reinserts, which fight against the dynamic behavior of R-tree indexing. Our idea is to substitute the usually employed MBRs with arbitrarily rotated ellipsoids. Intuitively, ellipsoid, as a quadratic surface, could *cover* the data more tightly, leading to regions smaller in volume (less dead space coverage) and possibly less overlaps (i.e., smaller regions naturally produce less overlaps). To grasp our "motivation", consider Figure 2a – 10 randomly generated tuples are covered by an ellipsoid and an MBR. The volume of the ellipsoid is 0.074 and the volume of the MBR is 0.181 so the ellipsoid is 2.4× smaller in volume than the MBR.

### 3.1 Index Structure

The structure of eR-tree is based on the original Guttman's R-tree (Guttman 1984). The most straightforward application of ellipsoids in R-tree would substitute all MBRs with ellipsoids. However, our preliminary tests revealed the following facts:

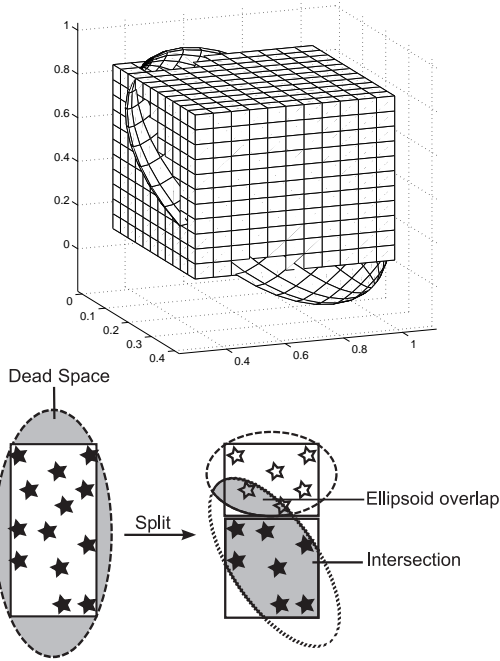


Figure 2: (a) Covering of 10 randomly generated tuples by ellipsoid and rectangle. (b) Dead space coverage and non-overlap-free split.

- The test for an ellipsoid and a query window (QW) intersection (employed in the query algorithm) is approx.  $50\times$  more expensive in terms of CPU costs than a test for MBR and QW intersection (for more details see Section 3.4).
- Most of the filtration/pruning is performed in nodes just one level above the leaf nodes; we call them *pre-leaf* nodes (and pre-leaf level).

Following the above observations and taking into account high storage requirements of ellipsoids (being quadratic with respect to the dimensionality), we decided to investigate the eR-tree variant with ellipsoids utilized only in pre-leaf nodes. The other nodes will use usual MBRs.

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

- When splitting, the ellipsoids tend to produce more overlaps than MBRs on dense data.
- When the data distribution tends to rectangular clusters, the ellipsoids will cover more dead space.

In Figure 2b, we can notice the effects stated above – an unnecessary dead space coverage on the left, and an overlap between ellipsoids on the right (after splitting). To solve these problems, we decided to define pre-leaf node region as an *intersection* of an ellipsoid and an MBR, consider Figure 3. A similar idea is engaged in SR-tree (Katayama & Satoh 1997), where regions are defined as an intersection of a sphere and an MBR.

### 3.2 Ellipsoid Theory

In this section we will discuss the problem of enclosing a set of data tuples by an ellipsoid.

**Definition 1** Let an ellipsoid  $\varepsilon(c, Q)$  in  $R^n$  with center in  $c \in R^n$  and shape matrix  $Q \in R^{n \times n}$  be the set of points

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

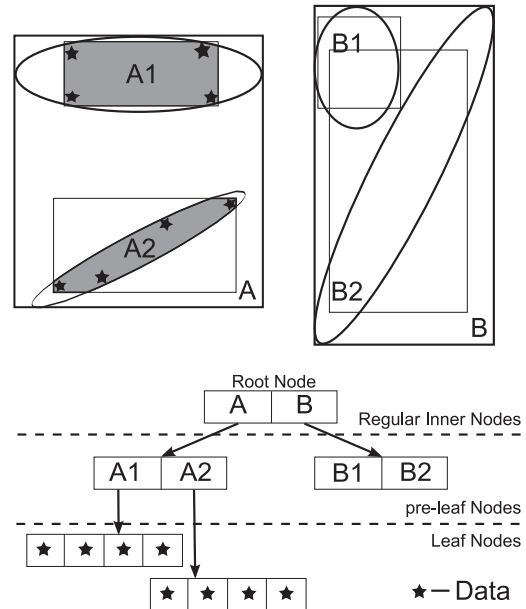


Figure 3: An eR-tree with pre-leaf regions defined as intersection of ellipsoid and MBR.

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

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

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

Further details and references on proof of the theorem could be found in (Sun & M. Freund 2004). It is evident to require an ellipsoid of minimum volume to be employed by eR-tree, hence, we define the minimum volume covering ellipsoid (MVCE) and outline the algorithm of MVCE construction with some performance experiments.

**Definition 2** For a given set  $S = \{x_1, \dots, x_k\}$  of  $n$ -dimensional tuples we define the Minimum Volume Covering Ellipsoid as any ellipsoid  $\varepsilon(c, Q)$  for which

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

Basically, we are aware of three distinct approaches how to construct MVCE. The first one published in early 80's is based on eigenvalue decomposition (Barnes 1982). Almost ten years later Welzl published an algorithm based on randomized iterative construction (Welzl 1991). Finally, Kchachiyani formulated the problem of computing MVCE as an optimization problem using interior-point method (Khachiyan & Todd 1993).

After careful examination of all methods of MVCE construction we decided to use the Kchachiyani optimization construction (Khachiyan & Todd 1993). The complexity of this algorithm was improved in (Todd & Yildirim 2007, Sun & M. Freund 2004, Kumar & Yildirim 2005). The latter brings the *Core sets*<sup>1</sup> as a byproduct. In this paper we will stick to MVCE construction described in (Moshtagh 2005).

<sup>1</sup>Core set is a small subset of the input tuples whose covering is "almost" same as the covering of the entire input, hence it can be used to optimize on large set of tuples.

We want to obtain the resulting ellipsoid of minimum volume and covering all the tuples from  $S$ . Thus, the formulation of MVCE is the following:

$$\begin{aligned} \text{minimize} \quad & \det(Q^{-1}) \\ \text{while preserving} \quad & \\ & (x_i - c)^T Q (x_i - c) \leq 1 \quad i = 1, \dots, |S| \\ & Q \succ 0 \end{aligned} \quad (1)$$

Since this problem is not a convex optimization problem, we can rewrite it, by substitution of  $A = Q^{1/2}$ ,  $b = Q^{1/2}c$ , as:

$$\begin{aligned} \text{minimize} \quad & \log(\det(A^{-1})) \\ \text{while preserving} \quad & \|Ax_i - b\| \leq 1 \quad i = 1, \dots, |S| \\ & A \succ 0 \end{aligned} \quad (2)$$

which is a convex optimization problem, but unfortunately still difficult to solve. Luckily, the dual problem is much easier. To solve the dual problem, *lifting* of the original problem needs to be defined. This means all tuples  $S = \{x_i, \dots, x_k\}$   $x_i \in R^n$  need to be moved to  $R^{n+1}$ . We can set  $(x_i^l)^T = [x_i^T, 1]$   $i = 1, \dots, k$  and define  $S^l = \{\pm x_1^l, \dots, \pm x_k^l\}$ . The  $mvce(S^l)$  will be symmetric around the origin of  $R^{n+1}$  and the  $mvce(S)$  will be obtained as an intersection of  $mvce(S^l)$  with the hyperplane  $H = \{(x, 1) \in R^{n+1} | x \in R^n\}$ . The lifted optimization is then

$$\begin{aligned} \text{minimize} \quad & \log(\det(M^{-1})) \\ \text{while preserving} \quad & (x_i^l)^T M x_i^l \leq 1 \quad i = 1, \dots, |S| \\ & M \succ 0 \end{aligned} \quad (3)$$

where  $M$  is the decision variable. Now, dual Lagrangian can be formulated and optimized. The optimization is carried out by Conditional Gradient Ascent method. The asymptotic complexity of this algorithm is *linear* in the number of tuples being covered by the ellipsoid. For further details we refer to (Moshtagh 2005) and (Kumar & Yildirim 2005). The former deals with details of optimization and solution extraction, in the latter the asymptotic complexity is derived.

As a *stopping criterion* of the optimization, the average distance of all tuples which lie outside the approximated ellipsoid to its boundary is used. While we need to have all the tuples strictly included in the resulting ellipsoid, we are performing a post-processing on the ellipsoid obtained from the approximation. In the post-processing, we locate the tuple which lies furthest from the boundary of the ellipsoid and then we scale the ellipsoid, so this tuple and all others lie inside. In the result, the larger stopping criterion, the faster computation of MVCE approximation but also larger volume. For example, a value 0.1 of the stopping criterion means the furthest tuple is at most 0.1 distant from the surface (considering unitary radius of an ellipsoid).

We conducted some small performance experiments to evaluate the adequacy of this constructing technique of MVCE for our purposes. In Table 1, we can observe the impact of the size of  $S$  and the dimensionality on the time needed to construct MVCE. The stopping criterion of the algorithm was set to 0.01.

The rest of experiments were focused on the stopping criterion, considering 3-dimensional space. In Table 2, we can notice the efficiency of the algorithm depends mostly on this parameter. The experimental results prove this algorithm meets the requirements for our purposes with stopping criterion at most 0.01.

SetSize	dim. 2	dim. 5	dim. 10	dim. 15	dim. 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

Table 1: Time (seconds) required to construct MVCE as a parameter of dimension (2, 5, 10, 15, 20) and set size.

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

Table 2: Time (seconds) required to construct MVCE as a parameter of stopping criterion (0.1, 0.01, 0.001) and set size.

### 3.3 Indexing

The algorithm of insertion is described in many available literature, e.g., the original (Guttman 1984), therefore we emphasize just our modifications. First, in Algorithm 1 see the procedure which locates the appropriate leaf for new entry accommodation.

---

#### Algorithm 1 ChooseLeaf

---

**Require:**  $P$  – tuple 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 current node, which region is closest to the  $P$ . The distance of ellipsoid and  $P$  is considered.}
- 5:   **else**
- 6:      $CurrentNode = getSubBranchMBR(P)$  {returns the child of current node, which MBR region being enlarged by  $P$  produces smallest enlargement. If enlargement should cause overlap with other regions of  $CurrentNode$  and there exists region of  $CurrentNode$  which won't produce overlap after being enlarged by  $P$ , than the sub-branch of this non-overlap-producing region is chosen. I.e. we prefer rather to cover more dead space, than to produce overlaps.}
- 7:   **end if**
- 8: **end while**
- 9: **return**  $CurrentNode$

---

The leaf splitting strategy significantly determines the performance of R-tree. In Algorithm 2, see our splitting algorithm, which tries to separate data tuples based on the dimension with the maximum variance. First, we choose the dimension in which the data is spread the most. Then we sort the data according to values in this dimension and find the split position.

---

#### Algorithm 2 MinVar Split

---

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

- 1:  $maxVar = 0.0$
- 2: **for**  $i = 0$  to  $dimension$  **do**
- 3:    $var = computeVarianceInDimension(i, \{p_1, \dots, p_k\})$
- 4:   **if**  $var > maxVar$  **then**
- 5:      $maxVar = var$ ;  $splitDimension = i$
- 6:   **end if**
- 7: **end for**
- 8:  $\{p'_1, \dots, p'_k\} = sortByDimension(splitDimension, \{p_1, \dots, p_k\})$   
    {so that  $\{p'_1 \leq p'_2 \leq \dots \leq p'_k\}$  holds in  $splitDimension$ }
- 9:  $diff = p'_k[splitDimension] - p'_1[splitDimension]$
- 10: **for**  $i = 2$  to  $k - 1$  **do**
- 11:   **if**  $p'_1[splitDimension] + diff/2 < p'_i[splitDimension]$  **then**
- 12:      $splitOrder = i$
- 13:     **break**
- 14:   **end if**
- 15: **end for**
- 16: **return**  $\{p'_1, \dots, p'_{splitOrder}\}, \{p'_{splitOrder+1}, \dots, p'_k\}$

---

### 3.4 Querying

The querying algorithms are also described in many available literature, e.g., in the original (Guttman 1984). Therefore, we will focus just on the ellipsoid and query window intersection test. The problem of deciding whether a query window  $QW(ql, qh)$ ,  $ql, qh \in R^n$  and an ellipsoid  $\varepsilon(c, Q)$  intersect, can be formalized as a convex optimization problem of form:

$$\begin{aligned} & \text{minimize} && (x - c)^T Q (x - c) && (4) \\ & \text{while preserving} && ql \leq x_i \leq qh_i && i = 1, \dots, n \end{aligned}$$

where the objective variable is  $x$ . If  $x \leq 1$ , then QW and ellipsoid intersect, otherwise their intersection is empty. We have conducted some experiments to compare the speed of intersection test of MBR×QW and Ellipsoid×QW, see Table 3. The optimization was

carried out with the *loqo*<sup>2</sup> solver. We can observe, that Ellipsoid×QW test is significantly slower than MBR×QW test, however, it is still significantly faster than a seek to secondary storage.

	dim.=2	=4	=6	=8	=10
MBR	0.000312	0.0005	0.000812	0.000686	0.000812
Ellipsoid	0.0403	0.0445	0.0458	0.0493	0.0602
	dim.=12	=14	=16	=18	=20
MBR	0.00112	0.00119	0.0012	0.00137	0.00194
Ellipsoid	0.0676	0.106	0.23	0.522	1.01

Table 3: Time (milliseconds) required to evaluate whether Ellipsoid/MBR and Query window intersect, depending on growing dimensionality.

## 4 Experimental Results

To evaluate the eR-tree, we have used two datasets:

**Clustered dataset** is a synthetic dataset and contains sets of  $1 \cdot 10^4 - 2 \cdot 10^6$  tuples, where each cluster is composed of 1,000 uniformly distributed tuples. Individual tuples were randomized prior to storing (so they are not indexed “cluster by cluster”, but randomly). We will refer to this dataset as to **SCU dataset**.

**Real dataset** was generated from the well-known IMDB database<sup>3</sup>. The 5-dim. tuples represent records [movie\_id, director\_id, movie\_genre\_id, movie\_production\_year, movie\_kind\_id]. The entire dataset consists of 392,689 records. The cardinality of individual attributes can be found in Table 4. We will refer to this dataset as to **IMDB dataset**.

Attribute	MIN	MAX
movie_id	1	1137185
director_id	31	1824450
movie_genre_id	2	29
movie_production_year	1519	2013
movie_kind_id	1	7

Table 4: Cardinality of individual attributes in the IMDB dataset.

We have generated various query sets to evaluate the eR-tree querying performance. Each query set consisted of 1,000 individual queries, and the performance results for one query set are presented as the average of all 1,000 trials. A query set is characterized by its *selectivity*, i.e. the number of hits each individual query returns. The querying tests are performed with query sets of absolute selectivity 3 and 50 tuples.

The eR-tree was implemented in C++, while all the experiments were carried out on 2.2GHz AMD Turion Processor with 1GB of RAM and 5400rpm hard drive on WinXP with NTFS file-system. The source code was compiled with VC 8.0.

For a comparison, the results for eR-tree are presented along with results for R-tree (being a baseline), however, we used an R-tree version extending the original Guttman’s version (Guttman 1984) by small improvements. These improvements try to avoid region overlaps at the cost of higher dead space coverage<sup>4</sup>, and relax the minimum required node utilization below 50%.

<sup>2</sup>Available at <http://www.princeton.edu/~rvdb/loqo/>

<sup>3</sup>Available at [www.imdb.com](http://www.imdb.com)

<sup>4</sup>As discussed in (Beckmann et al. 1990), the regions overlaps have greater impact on the search performance than the dead space

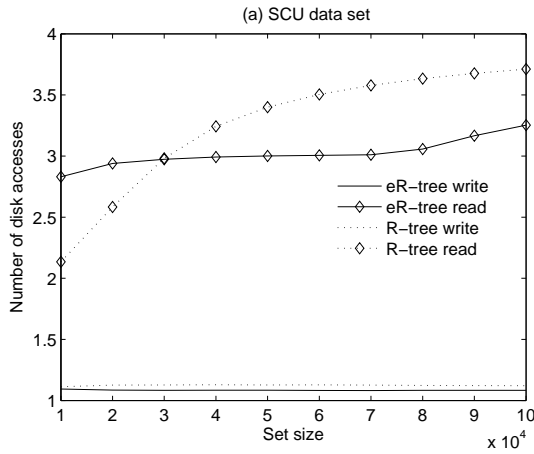


Figure 4: Insertion costs as a parameter of a set size – SCU dataset

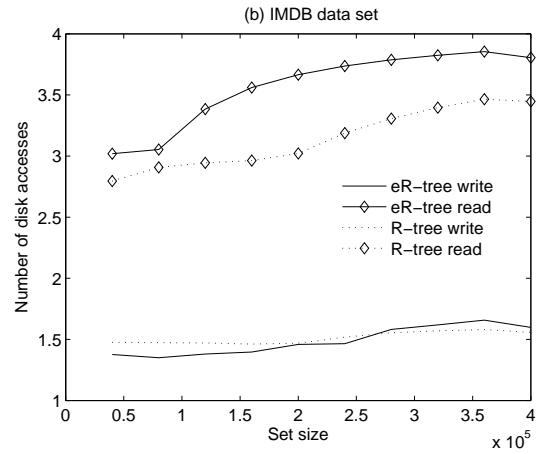


Figure 5: Insertion costs as a parameter of a set size – IMDB dataset

#### 4.1 Indexing

To meet the page size of the disk drive, the size of all the nodes of eR-tree and R-tree was set to 8KB. The fanout of eR-tree nodes can be found in Table 5. The heights of the trees for SCU and IMDB datasets can be found in Table 6. The columns marked with (\*) represent the theoretical tree heights with 100% utilized nodes.

Fanout	Dimension					
	3	4	5	8	10	12
inner node	146	113	93	60	48	40
pre-leaf node	39	26	19	9	6	4
leaf node	255	204	170	113	92	78

Table 5: Node fanout

Data set	Height			
	eR	R	eR(*)	R(*)
SCU	5	8	3	2
IMDB	3	3	3	2

Table 6: Tree heights

On the SCU dataset the eR-tree’s index size was 3.62 MB, compared to 5.2 MB for R-tree (average node utilization of eR-tree reached 69.7%, while R-tree reached only 50.2%). On the real IMDB dataset the eR-tree index size was 18.2 MB, compared to R-tree’s 15 MB (having 57.8% and 63.1% average node utilization, respectively).

In Figures 4, 5, see the insertion costs for a single tuple in terms of the number of required reads and writes. In Figure 4 the result for the SCU dataset with size varying from  $1 \cdot 10^4$  to  $1 \cdot 10^5$  is presented. In Figure 5, see an analogical experiment for the IMDB dataset.

Next, we have observed the insertion costs as a parameter of dimensionality. In Figure 6 see the measurements for the SCU dataset and Figure 7 for the IMDB dataset. For a more detailed comparison, see (Danko n.d.).

#### 4.2 Querying

In Figures 8-11, see the querying results for SCU and IMDB dataset, with respect to varying set size. On the left y-axis the total I/Os required to evaluate a

coverage; therefore, such an observation should slightly improve the performance of the original R-tree.

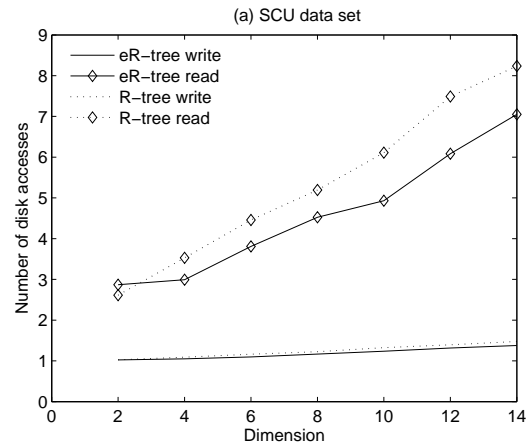


Figure 6: Insertion costs as a parameter of dimension: SCU dataset

query are plotted, on the right y-axis the number of searched leaf regions during evaluation is plotted. It can be seen, eR-tree on the SCU dataset clearly outperforms the R-tree – the eR-tree requires only 72% of I/O operation required by the R-tree. The difference is even more noticeable on query set with selectivity 50. On the IMDB dataset, the R-tree slightly outperforms eR-tree with selectivity of the query set equal to 3, however, with query set of selectivity 50 the eR-tree outperforms the R-tree significantly.

How about the impact of a dimensionality on the querying performance? This question is answered in Figures 12-15, where the SCU and IMDB datasets with varying dimensions are queried. We can notice, that eR-tree gains better results for lower dimensionalities (i.e., less than 10). In higher dimensions, the test for ellipsoid and QW intersection (recall, that it is an optimization problem) becomes more expensive, because there are “more” directions in which the optimization can go. To avoid these situations, if the intersection procedure needs 60 iterations or more, the optimization is stopped and the ellipsoid and QW are claimed to be intersected (to avoid false dismissals). As a consequence, the ellipsoid volume is overestimated (leading to more frequent intersections with QW), so extra leaf nodes need to be searched. In this paper we are not presenting the CPU time, because it could be misleading as it highly depends on a level of the intersection code optimization. However, using a general-purpose solver in our experiment, there were some situations where the eR-tree outperformed R-tree also in terms of CPU time (e.g., in low dimen-

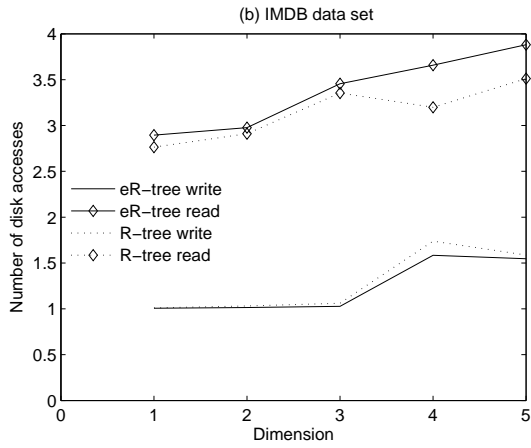


Figure 7: Insertion costs as a parameter of dimension: IMDB dataset

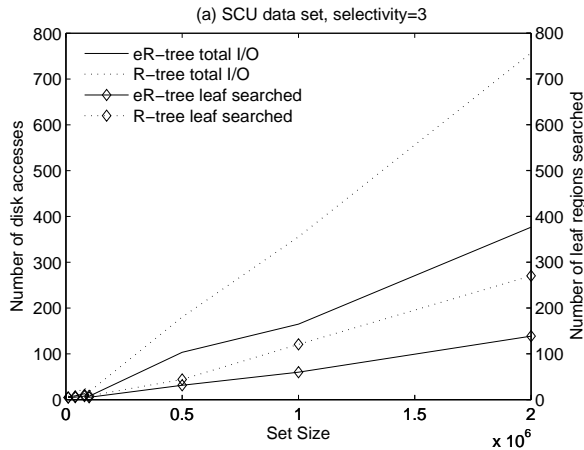


Figure 8: Querying cost as a parameter of a set size. SCU dataset, selectivity=3

sions on the IMDB data set).

## 5 Conclusions

In this paper, we have proposed the eR-tree, a variant of R-tree which employs minimum volume covering ellipsoids instead of usual minimum bounding rectangles. The experimental results shown that the construction of ellipsoids is efficient enough to be incorporated into R-tree-like index structures. 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. However, on dense data the advantage of ellipsoids is suppressed. In the future work the attention should be paid to optimization of the ellipsoid and query window intersection test and also the splitting algorithms. The further investigations should also favor of *sparse data*, where the ellipsoids outperform minimum bounding rectangles. In particular, besides native engines, XML databases are often transformed into sparsely distributed multi-dimensional tuples (Mlynkova & Pokorny 2008, Krátký et al. October 27-31, 2002). Here the eR-tree-based indexing could be beneficial for processing of an XPath or XQuery statement.

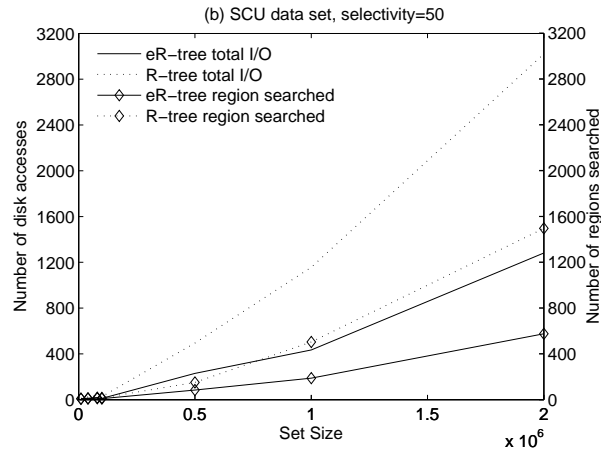


Figure 9: Querying cost as a parameter of a set size. SCU dataset, selectivity=50

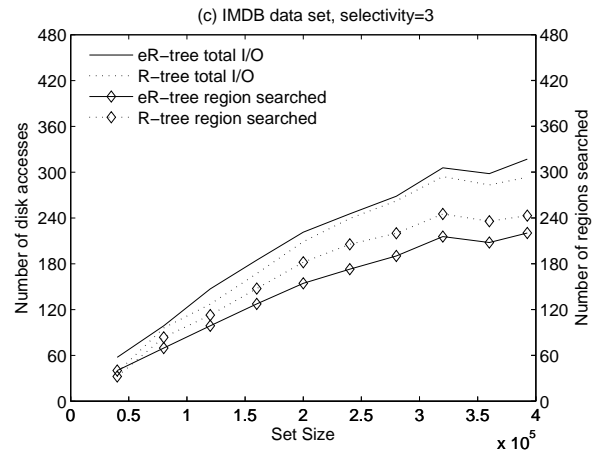


Figure 10: Querying cost as a parameter of a set size. IMDB dataset, selectivity=3

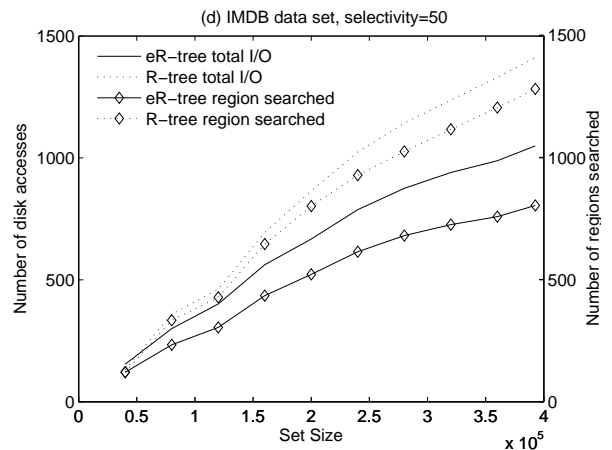


Figure 11: Querying cost as a parameter of a set size. IMDB dataset, selectivity=50

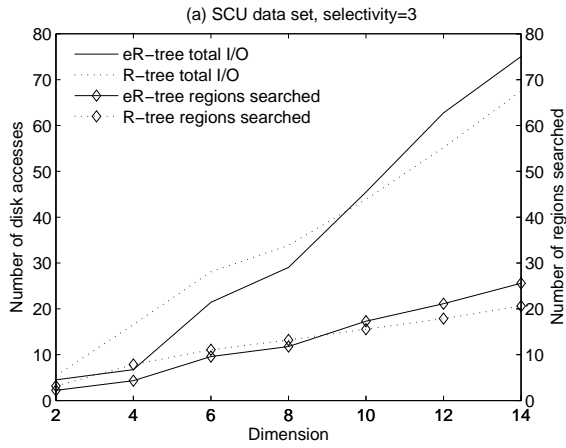


Figure 12: Querying cost as a parameter of a dimension. SCU dataset, selectivity=3

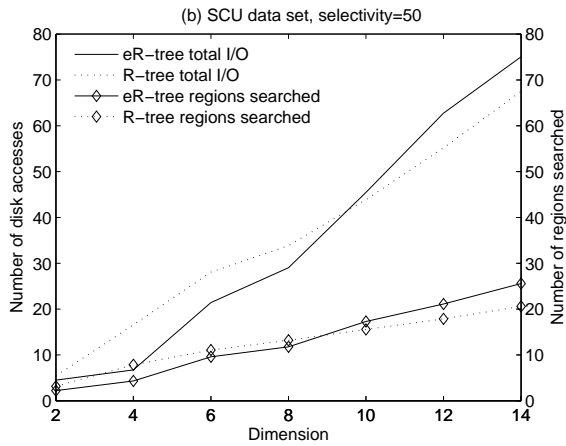


Figure 13: Querying cost as a parameter of a dimension. SCU dataset, selectivity=50

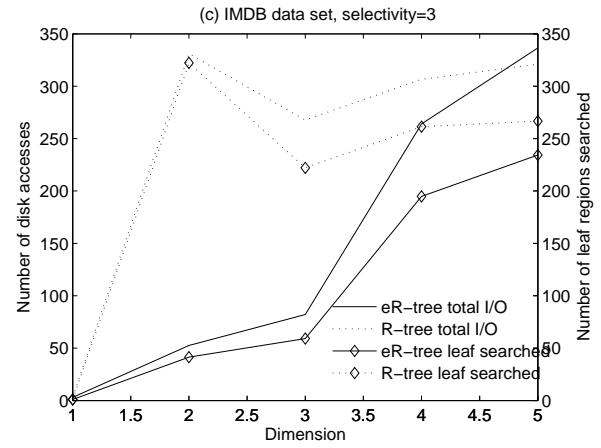


Figure 14: Querying cost as a parameter of a dimension. IMDB dataset, selectivity=3

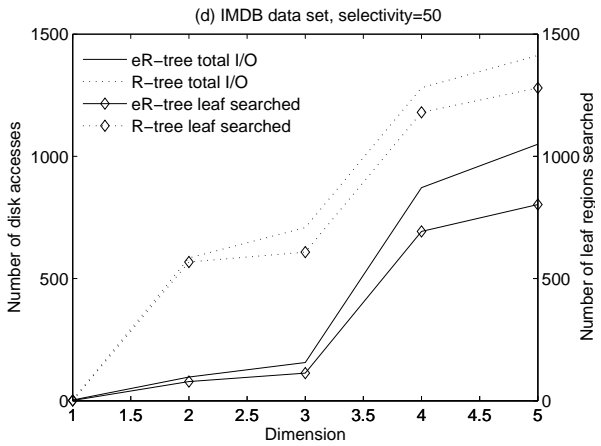


Figure 15: Querying cost as a parameter of a dimension. IMDB dataset, selectivity=50

## Acknowledgments

This research has been partially supported by Czech grants: GAČR 201/06/0756 and Institutional research plan number MSM0021620838.

## References

- Barnes, E. (1982), ‘An Algorithm for Separating Patterns by Ellipsoids’, *Image Processing and Pattern Recognition* **26**(6), 759.
- Beckmann, N., Kriegel, H.-P., Schneider, R. & Seeger, B. (1990), ‘The R\*-Tree: An Efficient and Robust Access Method for Points and Rectangles’, in H. Garcia-Molina & H. V. Jagadish, eds, ‘Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990’, ACM Press, pp. 322–331.
- Berchtold, S., Keim, D. A. & Kriegel, H.-P. (1996), ‘The x-tree: An index structure for high-dimensional data’, in T. M. Vijayaraman, A. P. Buchmann, C. Mohan & N. L. Sarda, eds, ‘VLDB’96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India’, Morgan Kaufmann, pp. 28–39.
- Danko, O. (n.d.), ‘Elliptic Indexing of Multidimensional Databases’, master thesis, Charles University in Prague,

[siret.ms.mff.cuni.cz/skopal/diplomky/danko.pdf](http://siret.ms.mff.cuni.cz/skopal/diplomky/danko.pdf), 2008’.

- Guttman, A. (1984), ‘R-Trees: A Dynamic Index Structure for Spatial Searching’, in B. Yomark, ed., ‘SIGMOD’84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984’, ACM Press, pp. 47–57.
- Katayama, N. & Satoh, S. (1997), ‘The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries’, in J. Peckham, ed., ‘SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA’, ACM Press, pp. 369–380.
- Khachiyan, L. G. & Todd, M. J. (1993), ‘On the complexity of approximating the maximal inscribed ellipsoid for a polytope’, *Mathematical Programming: Series A and B* **61**, 137 – 159.
- Krátký, M., Pokorný, J., Skopal, T. & Snášel, V. (October 27-31, 2002), ‘The Geometric Framework for Exact and Similarity Querying XML Data’, in ‘Proceedings of First EurAsian Conferences, EurAsia-ICT 2002, Shiraz, Iran’, Springer-Verlag LNCS 2510.
- Kumar, P. & Yildirim, E. A. (2005), ‘Approximate minimum volume enclosing ellipsoids using core sets’, *Journal of Optimization Theory and Applications* **1**, 1–21.

- Mlynkova, I. & Pokorny, J. (2008), Usermap : an adaptive enhancing of user-driven xml-to-relational mapping strategies, *in* A. Fekete & X. Lin, eds, 'Nineteenth Australasian Database Conference (ADC 2008)', Vol. 75 of *CRPIT*, ACS, Wollongong, NSW, Australia, pp. 165–174.
- Moshtagh, N. (2005), 'Minimum volume enclosing ellipsoid', *Convex Optimization* .
- Sellis, T. K., Roussopoulos, N. & Faloutsos, C. (1987), The R+-Tree: A Dynamic Index for Multi-Dimensional Objects, *in* 'VLDB', pp. 507–518.
- Sun, P. & Freund, R. (2004), 'Computation of Minimum Volume Covering Ellipsoids', *Discrete Applied Mathematics* **52**, 690–706.
- Todd, M. J. & Yildirim, E. A. (2007), 'On khachiyan's algorithm for the computation of minimum-volume enclosing ellipsoids', *Discrete Applied Mathematics* **155**, 1731–1744.
- Welzl, E. (1991), 'Smallest enclosing disks (balls and ellipsoids)', *Lecture Notes in Computer Science* pp. 359 – 370.
- White, D. & Jain., R. (1996), Similarity indexing with the SS-tree, *in* 'In Proc. 12th International Conference on Data Engineering (ICDE'96)', IEEE CS Press, pp. 516–523.