Benchmarking the UB-tree

Michal Krátký, Tomáš Skopal

Department of Computer Science, VŠB–Technical University of Ostrava,
tř. 17. listopadu 15, Ostrava, Czech Republic
michal.kratky@vsb.cz, tomas.skopal@vsb.cz

Abstract. In the area of multidimensional databases, the UB-tree represents a promising indexing structure. A key feature of any multidimensional indexing structure is its ability to effectively perform the range queries. In the case of UB-trees, we have proposed an advanced range query algorithm making possible to operate on indices of high dimensionality. In this paper we present experimental results of this range query algorithm.

Keywords: UB-tree, range query, benchmarks, DRU algorithm

1 Introduction

In multidimensional databases, objects are indexed according to several or many independent attributes. However, this task cannot be effectively realized using many standalone indices and thus special indexing structures have been developed is last two decades. Common to all this structures is that they index vectors of values instead of indexing single values.

The UB-tree represents one of the promising multidimensional index structures. Indexing and querying high-dimensional databases is a challenge for current research since high-dimensional indexing is significantly influenced by phenomenon called *curse of dimensionality*. This unpleasant phenomenon states that increasing dimensionality of feature space makes effective indexing and querying very hard (we refer to [2] and [8]). In this paper we present an advanced range query algorithm which makes the UB-tree suitable for indexing large high-dimensional databases.

In Section 1 we describe the UB-tree, Section 2 presents our range query algorithm and Section 3 analyses the comprehensive experimental results. Section 4 concludes the results.

1.1 Universal B-tree

The Universal B-tree (UB-tree) was introduced in [1] for indexing multidimensional data. Its main characteristics reside in an elegant combination of the well-known B^+-tree and the Z-ordering. The power of UB-tree lies in linear ordering of vectors, similarly like an ordering of simple values is indexed by the B^+-tree.

In the UB-tree we require to establish such ordering on a multidimensional vector space and thus linearize the space onto a single-dimensional interval which is usually realized using space filling curves [6]. A space filling curve orders all the points within a n-dimensional vector space. UB-tree was designed to be used with the Z-ordering generated using the Z-curve. Points (tuples) in the space are ordered according to their Z-addresses.

An interval $[\alpha : \beta]$ (α is the lower bound, β is the upper bound) on the Z-curve forms a region in the space which is called *Z-region*. An example of Z-curve and several Z-regions is presented in Figure 1a.

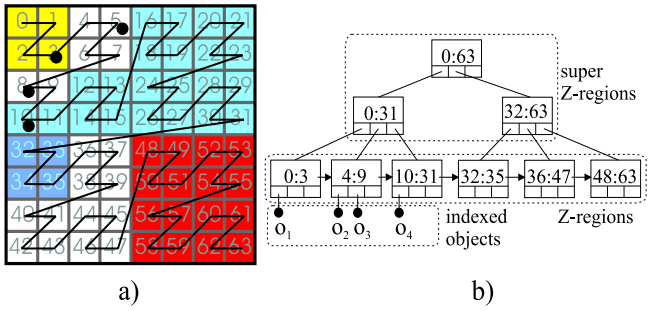

Fig. 1. a) The 2-dimensional space 8×8 filled with the Z-curve. The numbers in the grid are the Z-addresses. The space is partitioned with six Z-regions. The black dots represent 4 indexed objects. b) The UB-tree nodes correspond to the Z-regions and super Z-regions.

Each Z-region is then mapped into a single page within the underlying B^+-tree. The UB-tree leafs represent the Z-regions containing indexed objects themselves while the inner nodes represent the super Z-regions. A *super Z-region* contains all the (super) Z-regions lying entirely inside the super Z-region. Hence, the UB-tree structure is determined by a nested Z-region hierarchy. An indexed vector space and its appropriate UB-tree is depicted in Figure 1.

1.2 Range Queries

Realization of basic operations in the UB-tree (insertion, deletion, point query) is analogous to the operations in the "ordinary" B^+-tree. The main difference is that in the UB-tree we must at first compute the Z-address of the indexed object as a key for the subsequent operation on the underlying B^+-tree.

Unfortunately, a *range query* cannot be so simply forwarded to the B^+-tree. This fact arises from the speciality of the range query which is intended to be used on multidimensional data. Range query (window query respectively) in vector spaces is usually represented with a hyper-box in a given vector space Ω. The ranges of a query box QB are defined by two boundary points, the lower

bound $QB_{low} = [a_1, a_2, \ldots, a_n]$ and the upper bound $QB_{up} = [b_1, b_2, \ldots, b_n]$ where $a_1 \leq b_1, a_2 \leq b_2, \ldots, a_n \leq b_n$. The purpose of a range query is to return all the points located inside the query box, i.e. to return all the points o satisfying $a_i \leq o_i \leq b_i$, for $1 \leq i \leq n$ (see Figure 2a).

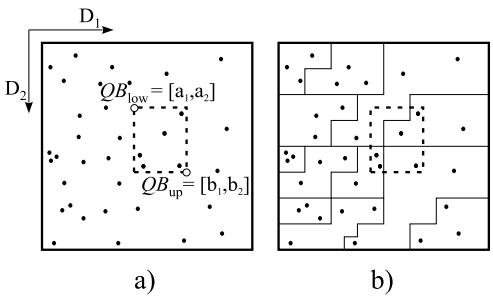

Fig. 2. a) Two-dimensional query box QB specified with lower bound QB_{low} and upper bound QB_{up}. b) Space Ω partitioned to Z-regions.

A more specific range query definition oriented to the UB-tree context can be formulated as a search over all the UB-tree's Z-regions that intersect given query box (see Figure 2b).

Existing Solution. Markl in [4] presents following range query algorithm consecutively searching intersecting Z-regions.

In Figure 3, an example of range query algorithm run is shown. At first, Z-address for the query box lower bound is computed. Using this value a page from UB-tree is retrieved and searched for relevant objects. Next, subsequent Z-region is retrieved and so on. The algorithm will finish as soon as the β of the active Z-region is greater than the Z-address of query box upper bound, i.e. $\beta > Zaddr(QB_{up})$.

So far, the algorithm was elegant and clear. But problem arises when we look deeper into function determining the next Z-address inside the query box. We denote this function `GetNextZaddress`. Computing the next Z-address lying within the query box is not trivial operation since this procedure is obviously dependent on the shape of Z-region. The algorithm for `GetNextZaddress` presented in [4] is of time complexity exponential with the dimensionality. Later, in [5], authors have presented a version linear with the Z-address bit length.

Limitations. Unfortunately, all descriptions of `GetNextZaddress` published so far were mentioned very briefly. Moreover, the explanations were always based on a pure algorithmic basis using "handling with bits" and hence lacking a geometric

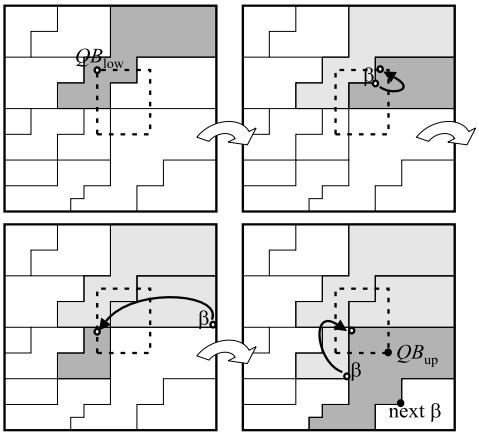

Fig. 3. Range query processing example

model providing a little bit deeper abstract view. Finally, original algorithms on UB-trees are protected with international patents[1].

2 Down-Right-Up Algorithm

The *DRU (Down-Right-Up) algorithm* exploits two types of leaf optimizations reducing unnecessary disk accesses as well as the Z-region intersection computations. The first optimization called *neighbour first point* is used for testing whether the first point of the right neighbour leaf (its Z-region respectively) lies inside the query box. If it does, the algorithm can simply "jump right" (the leafs are linked) to the neighbour leaf and continue processing. This optimization was already used in the original Bayer-Markl's algorithm.

The second optimization called *neighbour region* is specific to the DRU algorithm and is based on existence of `TestZregionIntersection` operation. This operation tests whether a given query box intersects a Z-region. We closely describe an algorithm realizing this operation as well as the theory to DRU algorithm itself in [7].

The *neighbour region* optimization is used for testing whether the neighbour leaf (its Z-region respectively) is intersecting the query box. If it does, the algorithm "jumps right" similarly like by the first optimization.

The DRU algorithm description:
The algorithm uses a *path stack* to keep the actual path in the UB-tree. The path stack allows us to avoid disk accesses to the nodes (and items in nodes) already processed.
DRU algorithm steps (input is the query box *qb*):

[1] Deutsches Patentamt Nr. 197 09 041.9 and Nr. 196 35 429.3

1. Find a leaf the Z-region of which contains Zaddr(qb_{low}). Store the path on the stack.
2. Search actual leaf for tuples lying inside qb. Return these tuples as a part of the result.
3. Retrieve the neighbour leaf from disk and set it as the actual leaf. If the first point of the actual leaf lies inside qb then goto step 2. This is the *neighbour first point* optimization.
4. If the Z-region of the actual leaf intersects qb goto step 2. This is the *neighbour region* optimization.
5. The stack must recover after the "optimization jumps". The UB-tree is passed (along the path in the stack) to the next relevant node. After the recovery, on the top of stack is a parent node of the leaf reached by the preceding optimization.
6. (**Right-Phase**). Peek the node on the top of the stack and try to find a link to the next relevant node (i.e. to node the Z-region of which intersects qb). If no such node is found, pop a node from the stack and repeat step 6 (**Up-Phase**). If a node is found, retrieve the node from the disk and push it onto the stack (**Down-Phase**). If a leaf is reached goto step 2 otherwise repeat step 6.

The algorithm terminates until a Z-region is found such that $\alpha \geq$ Zaddr(qb_{up}).

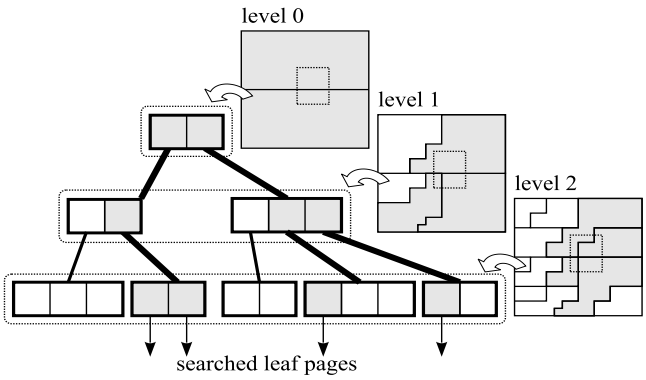

Fig. 4. DRU algorithm

An example of DRU algorithm is presented in Figure 4. Only the intersecting Z-regions (nodes respectively) are being processed. These Z-regions (nodes) are grayed. The bold branches are the only paths passed down. On the leaf level, all tuples lying inside the query box are returned as a query result.

3 Experimental Results

The tests were focused on several cost-factors. Besides the disk access costs (DAC), the effectivity of leaf optimizations was examined. Furthermore, computation costs (CC) of the range queries were investigated.

3.1 Cost Model

Let us now discuss the disk access costs and the computation costs. Let h be the height of the UB-tree, r be the number of Z-regions intersecting the query box, m_f be the number of neigbour leafs matched by the *neighbour first point* optimization, and m_r be the number of neighbour leafs matched by the *neighbour region* optimization.

The basic DAC $= (h + 1) \times r$. Had we consider the *neighbour first point* optimization, the DAC will be reduced to $(h+1)\times(r-m_f)+m_f$. Considering both leaf optimization will reduce the DAC to $(h+1)\times(r-(m_f+m_r))+(m_f+m_r)$.

The asymptotic CC $= \theta((h+2)\times r)$. Had we consider the *neighbour first point* optimization, the asymptotic CC will be reduced to $\theta((h+1)\times(r-m_f)+r)$. Considering both leaf optimizations will reduce the CC to $\theta((h+1)\times(r-m_f)-h\times m_r+r)$.

The set of tests was made on synthetic datasets of increasing dimensionality. The tuples were generated into randomly located clusters of fixed radius (using the L_2 metric) and indexed with the UB-tree. The number of tuples was increasing with the number of dimensions. In order to obtain solid results we have tested large datasets (up to 8 milion 30-dimensional tuples).

Query boxes of various shapes were generated randomly according to the distribution of tuples in the space. The ranges of query boxes were for growing dimensionality the same thus the volumes were increasing but the ratio query box volume/space volume was decreasing. This query box construction is typical for multidimensional applications. The number of queries was increasing with the number of dimensions (from 24 to 120 queries). The results are averaged.

The tests were performed on an Intel Pentium®4 2.4GHz with 512MB DDR333, 60GB UDMA100 7200rpm, run under Windows XP.

3.2 Two-Dimensional Datasets

For the two-dimensional datasets, we have examined the performance dependence on the growing UB-tree node capacity. In general, the greater node capacity implies lower disk access costs and number of computations.

UB-tree characteristics:			
$\|D_i\|$	2^{32}	dimensions	2
tuples	524,288	tree height	4–8
Z-regions	121,138–21,472		
node capacity	6–35	utilization	72.7–69.7%
node size	116–580B	index file	17.4–12.4MB
Range query characteristics:			
range queries	24	query selectivity (tuples)	9676.1
query real times	0.09–0.06 s		

Figure 5a shows the number of leaf Z-regions in two-dimensional UB-tree indices in order to node capacity. We can see that the growing leaf capacity "inflates" the Z-regions' volume while the number of Z-regions decreases. Figure 5b shows the number of Z-regions intersecting the query box. This indicates that the total volume of intersecting Z-regions is not dependent on the growing node capacity. In Figure 6, the disk access costs as well as the number of compu-

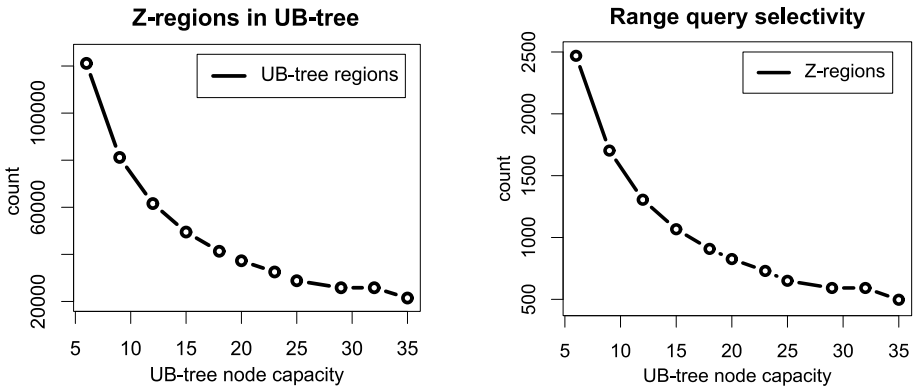

Fig. 5. a) Number of Z-regions. b) Number of Z-regions intersecting the query box.

tations for two-dimensional UB-tree indices are presented. The DRU algorithm performs significantly better than the original Bayer-Markl's algorithm since the DRU algorithms accesses by 30%-60% less disk pages. Similarly, the number of computations is lesser by 25%-50%.

The success of the DRU algorithm is caused by effective application of the *neighbour region* optimization. In Figure 7, the number of matching attempts of the leaf optimizations is presented. Matching attempt means a case when the Z-region *is* intersected, i.e. value *true* is returned and "jump" to the right neighbour leaf is performed. Figure 7a shows that the *neighbour first point* optimization is very effective for lower node capacities. The *neighbour region* optimization is performed after the *neighbour first point* was non-matching. Hence, the result is that for two-dimensional indices (low-dimensional respectively) the *neighbour first point* optimization filters the majority of unnecessary disk accesses. The third line shows the total attempts (matching and non-matching) of either optimization. In Figure 7b, the optimization effectivity is presented. The *neighbour*

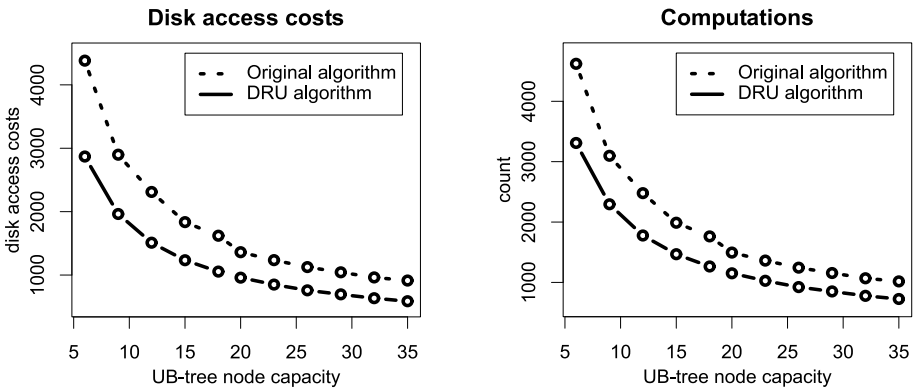

Fig. 6. a) Disk access costs. b) Number of computations.

first point optimization is effective by 90%, together with the *neighbour region* optimization by 95%.

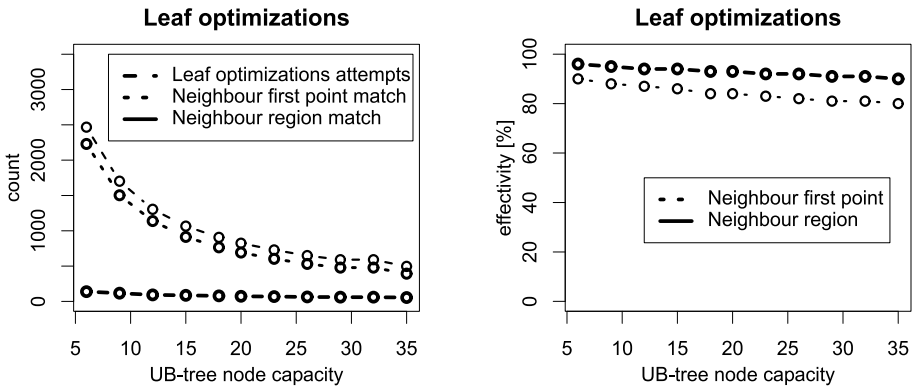

Fig. 7. a) Leaf optimizations. b) Leaf optimizations effectivity.

A particular result is that for low-dimensional UB-trees the Bayer-Markl's algorithm is only slightly less effective than the DRU algorithm. A different situation comes with more dimensions as we will see in the following sub-section.

3.3 High-Dimensional Datasets

In high-dimensional spaces, say for $n \geq 10$, the range query efforts rapidly increase. This fact is caused by the curse of dimensionality described later in this section. In practice, the disk access costs and the number of computations grow with the increasing dimensionality.

UB-tree characteristics:

$card(D)$	2^{32}	dimensions	2–30
tuples	524,288–7,864,320	tree height	4
nodes	22,400–321,885	Z-regions	21,475–321,885
node capacity	35	utilization	69.7–69.8%
node size	580–4612B	index file	12.4MB–1.44GB

Figure 8a shows the number of inserted tuples according to dimensionality of the dataset. In Figure 8b, the range query selectivity is depicted, i.e. average number of returned tuples. The other line in the graph represents the number of accessed leaf Z-regions. In Figure 9, the disk access costs and the number of

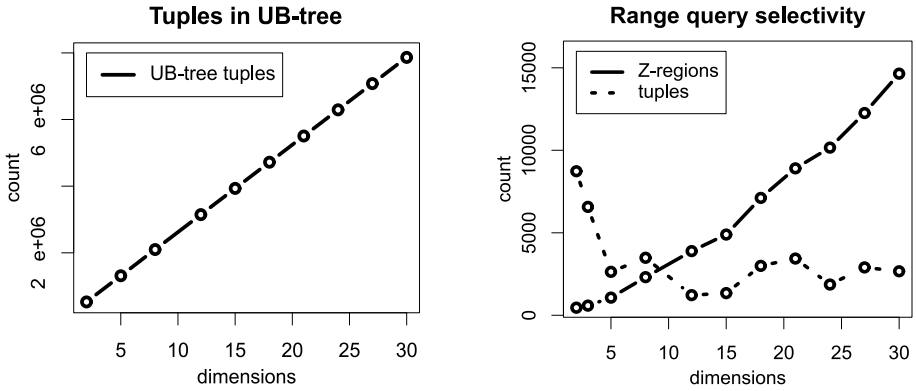

Fig. 8. a) Dataset sizes. b) Range query selectivity.

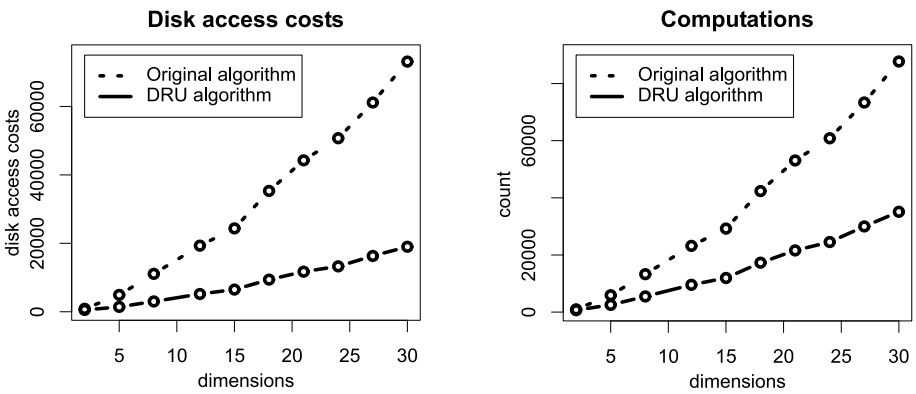

Fig. 9. a) Disk access costs. b) Computations.

computations are presented. We can see that with increasing dimensionality the costs grow. However, the growth for the DRU algorithm is much less steep than for the Bayer-Markl's algorithm. Thus, the DRU algorithm is more effective for

higher dimensionalities. The reason of the DRU algorithm's success resides again in the application of the leaf optimizations.

For higher dimensionalities, the significance of the *neighbour leaf* optimization exponentially grows while the *neighbour first point* optimization goes down to zero. This behaviour is caused by the complex shape of the Z-curve for higher dimensionalities thus the probability that the first point of the neighbour Z-region will intersect the query box tends to zero. On the other side, this kind of probability does not affect the effectivity of the *neighbour region* optimization since it determines the Z-region/query box intersection absolutely.

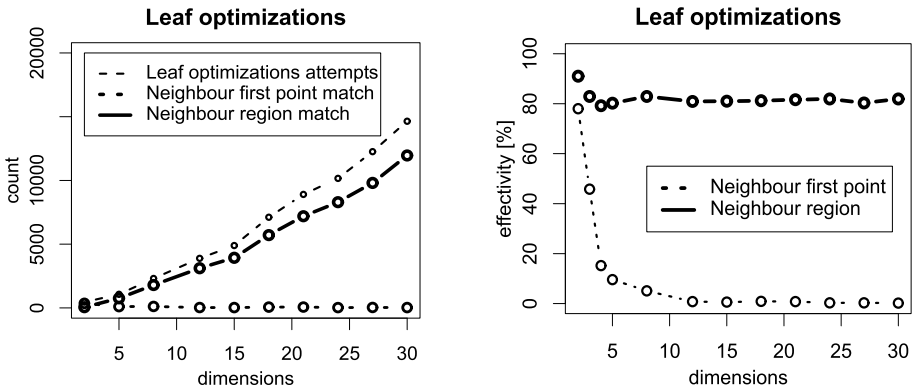

Fig. 10. a) Leaf optimizations. b) Leaf optimizations efficiency.

In Figure 11 we present average real times of a range query execution.

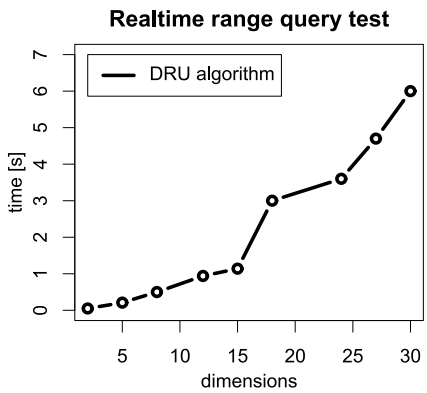

Fig. 11. Range query real times.

3.4 Curse of Dimensionality

Presented results allow us to think about the *curse of dimensionality* [2], [8] appearing in the UB-tree. With the growing dimensionality of UB-trees grow also the costs, even though less than exponentially. Figure 12a presents a ratio of tuples inside the query box to the number of intersecting Z-regions. Figure 12b shows ratio of intersecting Z-regions containing at least one tuple inside the query box to all of the intersecting Z-regions. This ratio says that in higher dimensionalities more than 95% of relevant Z-regions "give" no tuples to the result. The reason is obvious – the topological properties of the Z-curve are worse for higher dimensionalities.

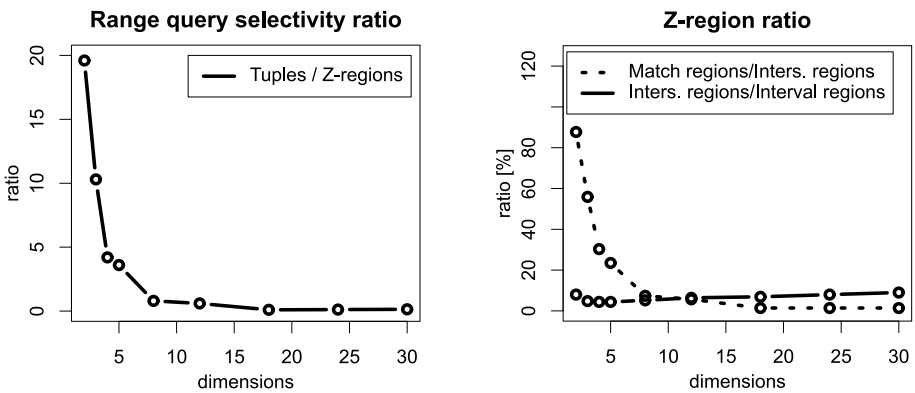

Fig. 12. a) Range query selectivity ratio. b) Query box ratios.

On the other side, the Figure 12b also shows a ratio of intersecting Z-regions to the Z-regions lying in the interval $[Zaddr(qb_{low}) : Zaddr(qb_{up})]$ (i.e. interval of the query box's "bounding Z-region"). One could expect that the negative effect of the curse of dimensionality will "raise" this ratio up to 100% which is the same as a traversal through the majority of the UB-tree structure. However, this test shows that (even for high dimensionalities) the number of Z-regions intersecting the query box is much lesser than the number of Z-regions within the above mentioned interval. This particular result indicates that the UB-tree together with the DRU algorithm is remarkably resistant to the curse of dimensionality. For a comparison, the well-known *R-tree* [3] used in many applications is very affected by the curse of dimensionality and its usage for high-dimensional indexing is nearly impossible.

4 Conclusions

The experimental results have shown that the DRU range query algorithm makes the UB-tree applicable for effective indexing and querying of high-dimensional feature spaces.

The key to the DRU algorithm effectivity is an incorporation of two leaf optimizations. The *neighbour region* optimization allows the DRU algorithm process range queries in high-dimensional spaces and thus proves that the UB-tree is partially resistable to the unpleasant curse of dimensionality. In particular, the DRU algorithm allows to effectively process "tight" range queries, i.e. query boxes having very disproportionate ranges.

References

1. R. Bayer. The Universal B-Tree for multidimensional indexing: General Concepts. In *Proceedings of World-Wide Computing and its Applications'97, WWCA'97, Tsukuba, Japan*, 1997.
2. C. Böhm, S. Berchtold, and D. Keim. Searching in High-Dimensional Spaces – Index Structures for Improving the Performance of Multimedia Databases. *ACM Computing Surveys*, 33(3):322–373, 2001.
3. A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of ACM SIGMOD 1984, Annual Meeting, Boston, USA*, pages 47–57. ACM Press, June 1984.
4. V. Markl, F. Ramsak, and R. Bayer. Improving OLAP Performance by Multidimensional Hierarchical Clustering. In *Proceedings of IDEAS Conference, Montreal, Canada*, 1999.
5. F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer. Integrating the UB-tree into a Database System Kernel. In *Proc. Of the 26th Int. Conference VLDB, Cairo, Egypt*, 2000.
6. H. Sagan. *Space-Filling Curves.* Springer–Verlag, 1994.
7. T. Skopal, M. Krátký, V. Snášel, and J. Pokorný. On Range Queries in Universal B-tree. In *submitted to VLDB 2003*, 2003.
8. C. Yu. *High-Dimensional Indexing.* Springer–Verlag, LNCS 2341, 2002.