ARG Technical Report

ARG-TR-01-2003

On Range Queries in Universal B-trees

Tomáš Skopal
Michal Krátký
Václav Snášel
Jaroslav Pokorný

On Range Queries in Universal B-trees

Tomáš Skopal, Michal Krátký, Václav Snášel
Department of Computer Science,
VŠB-Technical University of Ostrava,
Czech Republic
{tomas.skopal, michal.kratky, vaclav.snasel}@vsb.cz

Jaroslav Pokorný
Department of Software Engineering,
Charles University, Prague,
Czech Republic
pokorny@ksi.ms.mff.cuni.cz

Abstract

The Universal B-tree is a data structure for multidimensional data indexing. In this paper we introduce a new algorithm of processing the range query in UB-trees. This algorithm performs very well, especially in high-dimensional indexes and thus significantly reduces influence of the "curse of dimensionality". We explain the algorithm using a geometric model giving the reader understandable insight into the problem. The geometric model is based on an interesting fact that there exists a relation between the Z-curve and generalized quad-trees. We also present promising experimental results of our range query implementation.

1 Introduction

Indexing and querying data, that is the state of the art in the area of database systems. Data can be indexed according to a single attribute, e.g. with the B-tree. On the other side, we intend to index some data according to several or many independent attributes. We call this data *multi-dimensional* since an instance of that data is indexed as a vector of simple values. Collection of such vectors can be interpreted as a set of points within a multidimensional vector space. Methods allowing indexing and querying multidimensional data are called *Spatial Access Methods* (SAM). For a synoptic survey over various SAM we refer to [5].

In this paper we introduce a new range query algorithm for the UB-tree. This algorithm performs very well, especially in large high-dimensional indexes. In Section 1, the problem is described and existing solution is discussed. Our algorithm is presented in Section 2. Geometric principles behind the algorithm are elaborated in Section 3. Sections 4 and 5 analyse and conclude the experimental results.

1.1 Vector Spaces

Let us specify some necessary notations.

Definition 1 (vector space)
A *discrete vector space* Ω is determined as a Cartesian product of finite domains D_i, $\Omega = D_1 \times D_2 \times \ldots \times D_n$. Every vector space has n dimensions. Each particular domain D_i is associated with the i-th dimension of the space. A *point* (tuple) of the space $o \in \Omega$ is represented with a vector of coordinates $o = [o_1, o_2, \ldots, o_n]$ where $o_i \in D_i$.

We will assume for simplicity that the vector space Ω is a hyper-cube determined as the n-th power of a single domain D, i.e. $\Omega = D^n$, where D is a linearly ordered interval of natural numbers $D = \langle 0, 2^p - 1 \rangle$ thus the cardinality of D is $card(D) = 2^p$ for some p.

1.2 General Concepts of the UB-trees

The Universal B-tree (UB-tree) was introduced in [1] for indexing multidimensional data (a modification called *bounding UB-tree* (BUB-tree) was recently introduced [4], allowing to avoid indexing of the "dead space"). Its main characteristics reside in an elegant combination of the well-known B^+-tree and the Z-ordering. The power of B-trees lies in ordering keys indexed by this data structure. 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. How to linearize a vector space? Usually, space filling curves are used [12]. A space filling curve orders all the points within a n-dimensional vector space. In other words, each point in the space can be assigned to an *address*. This address is a unique number defining order of the point on the curve. This ordering is then combined with the B^+-tree. 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.

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 tuples themselves while the in-

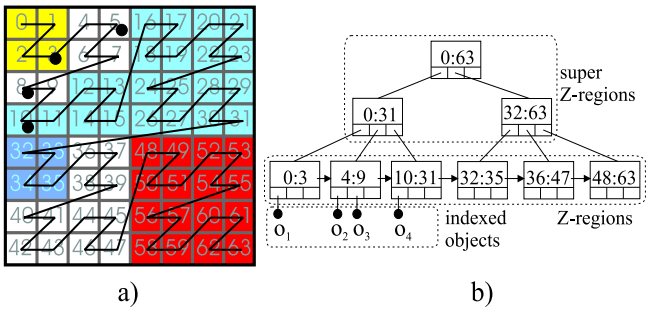

Figure 1. a) 2D space 8×8 filled with the Z-curve, partitioned with six Z-regions. b) The UB-tree nodes correspond to the Z-regions and super Z-regions.

ner 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.

1.3 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 tuple 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 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 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 tuples located inside the query box, i.e. to return all the tuples o satisfying $a_i \leq o_i \leq b_i$, for $1 \leq i \leq n$ (see Figure 2a).

Range query definition oriented to the UB-tree context can be formulated as a search over all the Z-regions intersecting given query box (see Figure 2b). For an overview on searching in high-dimensional spaces we refer to [2].

1.4 Existing Solution

Markl in [10] presents following range query algorithm consecutively searching intersecting Z-regions.

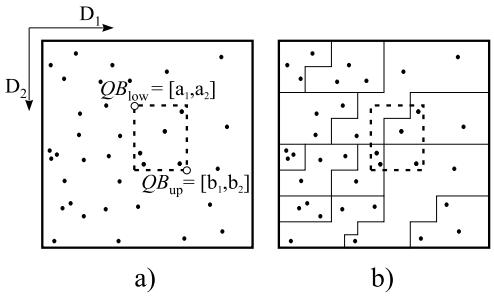

Figure 2. a) 2D query box QB specified with lower bound QB_{low} and upper bound QB_{up}. b) Space Ω partitioned to Z-regions.

1. Z-address of the query box's lower bound is computed, i.e. $Zval = Zaddr(QB_{low})$.
2. Repeat following steps while the $Zval$ is lower or equal than Z-address of the query box upper bound, i.e. while $Zval \leq Zaddr(QB_{up})$
   (a) At the deepest UB-tree level a page P is retrieved, the Z-region of which contains $Zval$, i.e. $\alpha \leq Zval \leq \beta$.
   (b) Page P is searched for all the tuples lying inside QB. These tuples go to the output as a part of the result.
   (c) Next intersecting Z-region must be determined. The first Z-address greater than β and intersecting the query box is found, i.e. a function `GetNextZaddress(`β`, `QB`)` is called, and the result is assigned to $Zval$.

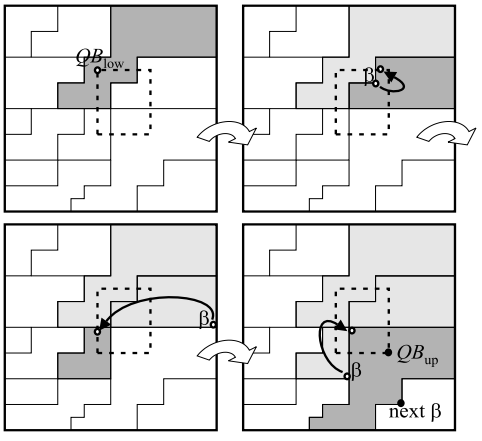

Figure 3. Bayer-Markl's range query algorithm.

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 tuples. 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 a problem arises when we look deeper into the 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 [10] is of exponential time complexity according to the dimensionality. Later, in [11], authors presented a version that is of linear complexity according to the Z-address bit length.

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

1.4.2 Time Complexity

Let h be the height of the UB-tree, m be the number of tuples stored in the UB-tree, and $c \geq 2$ be a fixed node capacity (arity of the UB-tree), i.e. $log_c(m) - 1 \leq h \leq log_2(m) - 1$. Let k be the number of Z-regions intersecting the query box. Complexity of the `GetNextZaddress` is linear according to the Z-address bit length, i.e. $O(n * log(card(D)))$.

In each step, the algorithm retrieves the next of the k intersecting Z-regions. This operation consists of one calculation of the `GetNextZaddress` and of one UB-tree downward pass (point query respectively) required for the Z-region retrieval. Thus, the overall time complexity of the range query is

$$O(k * h * n * log(c) * log(card(D)))$$

1.4.3 Disk Access Costs

The number of I/O operations performed during an algorithm run has an important impact on the algorithm efficiency. In the case of range query algorithm, the I/O operation is represented as a disk page retrieval. During the presented algorithm run, each Z-region is retrieved using h disk accesses. This retrieval is performed k times, thus the disk access costs factor is $DAC = k * h$.

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

1.5 Other Related Work

Almost the same algorithm for querying multidimensional data is presented by Lawder and King in [9, 8], but they have not applied the UB-tree as a framework. They have applied two types of space filling curves on this algorithm. In addition to the Z-curve they have mainly studied the Hilbert curve. In consequence, only the particular algorithm of appropriate `GetNextCurveAddress` operation was modified for the Hilbert curve. Alas, the problem of `GetNextZAddress` still remains, also these works explain it very vaguely.

2 Our Approach

We have focused on the basic straightforward idea that range query processing must search only Z-regions intersecting the query box. This can be realized via a single UB-tree downward pass. The UB-tree is passed in LIFO (last-in-first-out) fashion while each visited node is examined whether the Z-regions of its child nodes intersect the query box or not. Only intersecting nodes are further processed. On the leaf level, all tuples lying inside the query box are returned as a query result. The situation is outlined in Figure 4.

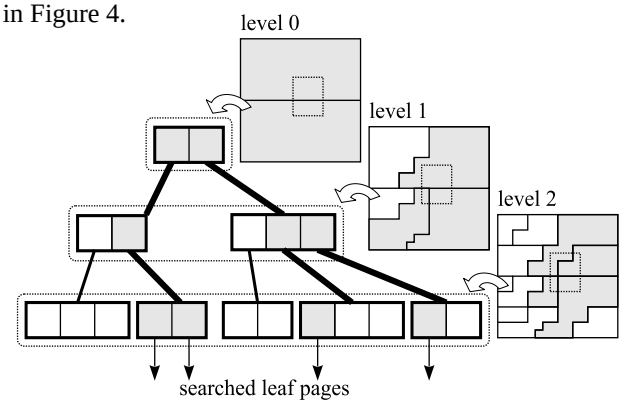

Figure 4. Single-pass range query algorithm. Only the intersecting Z-regions (nodes respectively) are being processed. These Z-regions (nodes) are grayed. The bold branches are the only paths passed down.

Similarly to the existing solution, our algorithm is also conditioned by a specific crucial operation. This particular operation relies on testing whether a given Z-region is intersecting the query box or is not. We will closely analyze this operation in Section 3.

2.1 Single-pass Algorithm Description

Algorithm 1 works according to the scheme outlined in Figure 4 and passes the UB-tree in LIFO fashion.

Algorithm 1 (single-pass range query)

```
RangeQuery(UBTree tree, HyperBox qb)
{
  recRangeQuery(tree.rootNode, qb, false)
}
recRangeQuery(UBNode node, Hbox qb, bool sFlag)
{
  If (Not node.Leaf) Then {
    If (Not sFlag) Then {
      // restrict to relevant child nodes
      start = GetStartNode(node, Zaddr(qb.lower))
      end = GetEndNode(node, Zaddr(qb.upper)))
    } Else {
      start = 0
      end = node.Capacity - 1
    }
    If (TupleInsideBox(node[start].Zregion.alpha, qb)) Then
      node[start].flag = true
    // test only relevant child nodes
    For i = start To end {
      // test Z-region intersection with qb
      If (node[i].flag Or TestZRegionIntersection(
                          node[i].Zregion, qb)) Then {
        If (i>start And i<end) Then sFlag = true
        Else sFlag = false
        // process the child nodes
        recRangeQuery(node.ReadChildNode(i), qb, sFlag)
        // minimizing intersection computations
        If (TupleInsideBox(node[i].Zregion.beta+1, qb)) Then
          node[i+1].flag = true
      }
    }
  }
  Else AddTuplesToResult(FilterTuplesWithBox(node.data, qb))
}
```

Notes to the pseudo-code:

- `GetStartNode(node, qb)` computes the index of a child Z-region in `node` with greatest α smaller than the Z-address of query box lower bound, i.e. searching `index` satisfying `node[index].alpha` $\leq$ `Zaddr(qb.lower)` And `node[index+1].alpha > Zaddr(qb.lower)`. `GetEndNode(node, qb)` does the same dually.

- To prune some `TestZRegionIntersection` executions, the `node[i].flag` for each Z-region is used. After the processing of `node[i]` is finished, the Z-address `node[i].Zregion.beta+1` is examined (which is actually the first point in the next Z-region, i.e. `node[i+1].Zregion.alpha`) and if the Z-address lies inside the query box, the Z-region is certainly intersecting and the `node[i].flag` is set to *true*. Thus the `TestZRegionIntersection` can be skipped.

- `TestZRegionIntersection(Zregion, qb)` evaluates whether a given Z-region is intersecting (overlapping) the query box. For further details and the algorithm see Sections 3.2 and 3.3.

- `TupleInsideBox(Zaddress, qb)` is used to examine whether the `Zaddress` lies inside the query box. This operation works with Z-address representations thus it does not require any conversions from/to Cartesian system (see [10]). Its complexity is linear according to the Z-address bit length.

- `FilterTuplesWithBox(Tuple[], qb)` returns tuples lying inside the query box.

The Algorithm 1 presented was designed to be simple and understandable. We have developed also an advanced version called *DRU algorithm* which significantly reduces the vertical UB-tree passing and thus minimizes disk access costs as well as the intersection computations. This improvement was feasible only due to the existence of the `TestZRegionIntersection` operation. We specify the DRU algorithm in Section 4 for understanding the experimental results.

2.2 Time Complexity

Complexity of the `TestZRegionIntersection` is linear according to the Z-address bit length (see Section 3.3), i.e. $O(n * log(card(D)))$. Had we compare the `TestZRegionIntersection` and (as declared) the `GetNextZaddress` operations, it is important to realize that they are of the same complexity as is the complexity of simple comparison of two Z-addresses. In other words, they are very cheap.

The algorithm retrieves k intersecting Z-regions by single LIFO pass. In each visited node, the `TestZRegionIntersection` is called at most $log_2(c)$-times (using halving the interval). Thus, the overall range query time complexity is the same as by the Bayer-Markl's alg.,

$$O(k * h * n * log(c) * log(card(D)))$$

2.3 Disk Access Costs

In Algorithm 1, each intersecting node is retrieved only once, which is a consequence of the single pass through the UB-tree. The DAC factor is $DAC \leq k * h$.

3 Z-region Intersection

In this section we will discuss some properties of the Z-curve which will help us to understand the shape of a Z-region. Using this knowledge we can design an algorithm for testing intersection between a Z-region and the query box.

3.1 Geometric Properties of the Z-curve

We can easily say that a space filling curve defines on a discrete space Ω a linear order of all the points within the

space. Three space filling curves are depicted in Figure 5. Note that a curve is only geometric interpretation of a linear order of points within a space. For more details about space filling curves we refer to the monograph [12].

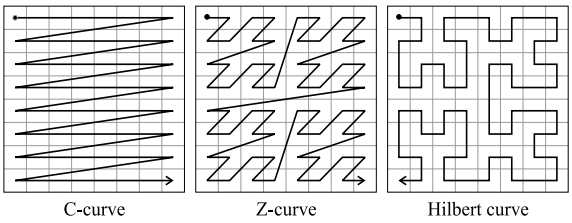

Figure 5. Space filling curves.

For the basic UB-tree operations (insertion, deletion, point query) there is no reason to choose the Z-curve or another specific space filling curve. The choice of the Z-curve for the UB-tree was made due to the easier implementation of range query along with the fast Z-address construction algorithm.

The important property of the Z-curve is its high *locality*. The locality concept in the case of Z-curve says that points that are "close" in the space (using some metric) are also "close" on the Z-curve (using the Z-order). In other words, the Z-curve carries topological information about the space – it localy preserves metric (we refer to [6]). Alternatively, Markl classifies space filling curves according to their *symmetry* (see [10]) – a self-similarity concept taken from fractal geometry. For key retrieval using fractals see [3].

Thus, to understand the range query in the UB-tree we must at first understand the nature of the Z-curve.

3.1.1 Z-address Construction

The simplest analytic description of the Z-address function is the following definition.

Definition 2 (Z-address)
Let us have an n-dimensional point $o \in \Omega$ where binary representation of each coordinate o_i is denoted as $o_i = o_{i,s-1}o_{i,s-2}\ldots o_{i,0}$. Then

$$Zaddr(o) = \sum_{j=0}^{s-1}\sum_{i=1}^{n} o_{i,j}2^{jn+i-1}$$

is called Z-address of o.

Anyhow simple this formula looks like, its imagination value seems to be quite low. A little bit more information about the Z-address construction is provided by the *bit interleaving* algorithm. Using this algorithm, the Z-address is constructed from the coordinates, where in each step the bits are "sliced" from the coordinates (one bit from each coordinate in each step) and "glued together" into a single bit string. The bit interleaving is depicted in Figure 6.

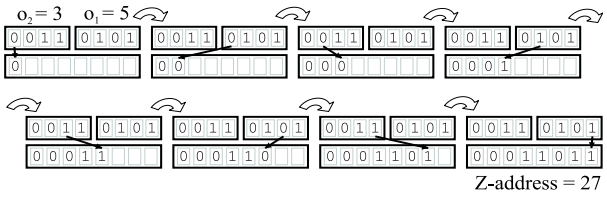

Figure 6. Bit interleaving algorithm for two-dimensional point $o = [5, 3]$.

We have tried to investigate some more characteristics about the Z-curve shape, especially those usable for the range query purposes. We took advantage of generalized quad-trees as the geometric framework.

3.1.2 Hyper-Quad Trees

Generalized quad-trees and their modifications were applied many times in the areas of CAD and GIS as well as in the area of SAM (see e.g. [13]). However, in our approach we need to consider the generalized quad-tree a little bit more abstractly since we will use it just as a formal tool for studying the Z-curve.

In geometry, the term *quadrant* does reflect an exactly defined quarter of a two-dimensional space. Similarly, we can divide the one-dimensional space on two halfs and in $three$-dimensional space we distinguish eight octants of a space. Common to all these geometric constructs is a need to *partition* the space. This partition is always realized using halving the space in all existing dimensions. We can generalize such knowledge through definition of a hyper-quadrant of an n-dimensional space.

Definition 3 (hyper-quadrant)
Let us have a vector space $\Omega = D^n$. Then *hyper-quadrant* (hquad) HQ is a subspace in Ω, i.e. HQ $\subset \Omega$, such that HQ = $\text{HD}_1 \times \text{HD}_2 \times \ldots \times \text{HD}_n$ where each domain HD_i is the lower or the upper half of the domain D, i.e. $\text{HD}_i = low(D)$ or $\text{HD}_i = up(D)$.

Corollary 1
Each n-dimensional vector space Ω is formed by its 2^n disjunct hquads.

However, the previous "set definition" does not handle with any identification of an hquad according to the location in the space. Actually, we can establish up to $(2^n)!$ orderings on the set of all hquads. Fortunately, there exists one suitable hquad numbering (ordering) that will help us

to discover a relation between the hyper-quad trees and the Z-curve. The desired hquad number – we call it *hquad code* – is constructed using successive halving of each dimension and testing whether the hquad is located in the lower or in the upper half of the dimension.

Definition 4 (hquad code)
The *hquad code* is represented as a binary string where each bit indicates the hquad location relatively to one dimension. The i-th bit is set to `0` if the hquad is located (relative to the i-th dimension) in the lower half of domain D, i.e. if $\mathrm{HD}_i = low(D)$. The second case is dual, i.e. the i-th bit is set to `1` if $\mathrm{HD}_i = up(D)$.

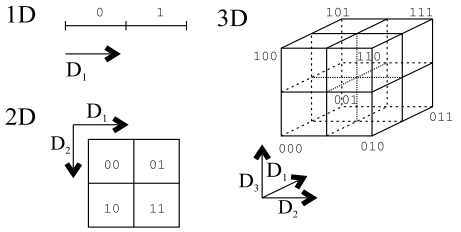

Figure 7. Hyper-quadrants and their codes.

The most significant bit (the left-most) indicates the location within the n-th dimension while the less significant bit (the right-most) indicates the location within the first dimension. If all the hquad codes are sorted lexicographically (left-to-right) we get the desirable ordering of hquads.

In Figure 7 the hquads and their codes are depicted. The bit length of the hquad code is n which is obvious from the code construction.

Definition 5 (HQ-tree)
Hyper-quad tree (HQ-tree) is generalized quad-tree. HQ-tree represents a simple hierarchical partition of an n-dimensional vector space. Each inner node of the HQ-tree contains a covering hquad and 2^n links to all its sub-hquads. Covering hquad of the root node is the whole space Ω while the leafs are all the points of Ω. The links to the sub-hquads are stored in ascending order according to their hquad codes.

Corollary 2
Because the domain D of a given vector space Ω is a finite set, height of the HQ-tree is $log_2(card(D))$. Hquads in the leafs of HQ-tree are points, i.e. the domains of the deepest hquads contain single element. Arbitrary point of the space Ω is always hquad located in a leaf in the appropriate HQ-tree.

Examples for 1D, 2D and 3D spaces are the binary tree, the quad-tree, and the octant-tree, see Figure 8.

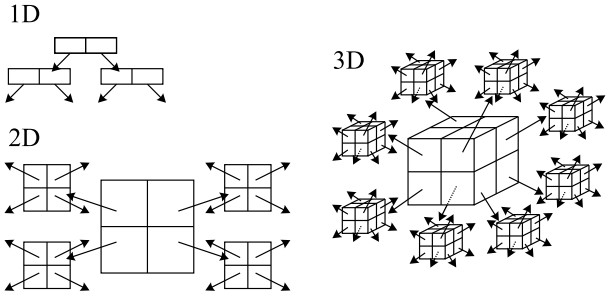

Figure 8. HQ-trees – binary tree, quad-tree, and octant-tree.

Finally, we need to establish unique identification of a hquad within the HQ-tree hierarchy. This we can manage using the downward *navigation pass* through the HQ-tree. During this pass *hquad navigation code* is constructed. This code of variable length uniquely determines the hquad position in the HQ-tree.

Definition 6 (hquad navigation code)
Let us have an hquad *hq* $\subset \Omega$. Then binary string $navi(hq) = c_0 \cdot c_1 \cdots c_{l-1}$ is called *hquad navigation code* if each substring c_i is a code of hquad (on the i-th level of the HQ-tree) that spatially contains *hq*.

Corollary 3
The larger hquads have shorter navigation codes and vice versa. The bit length of a point navigation code is $|navi(pt)| = n * log_2(card(D))$.
Figure 9 shows the navigation code construction. In the following we will present the relation between HQ-trees and the Z-curve we have earlier announced.

Proposition 1
Navigation code of $o \in \Omega$ is equivalent to the Z-address of o, i.e. $navi(o) = Zaddr(o)$.

Proof idea: If we realize, both of the Z-address construction algorithms, i.e. navigation code construction and the bit interleaving, do the same thing. However, the navigation code construction does it in terms of hquad codes concatenation. □

Another relation between the Z-curve and the HQ-trees is hidden in the LIFO HQ-tree pass. This pass will visit the HQ-tree leafs (points in space) in the same order as by filling the space with the Z-curve. Hence, we could sketch simple recursive algorithm of drawing the Z-curve using the HQ-tree pass. As a side effect, the "drawing algorithm" could serve as an effective tool for Z-ordering the *whole universe* Ω. This could be useful in various computer science disciplines where every point of a space carries some

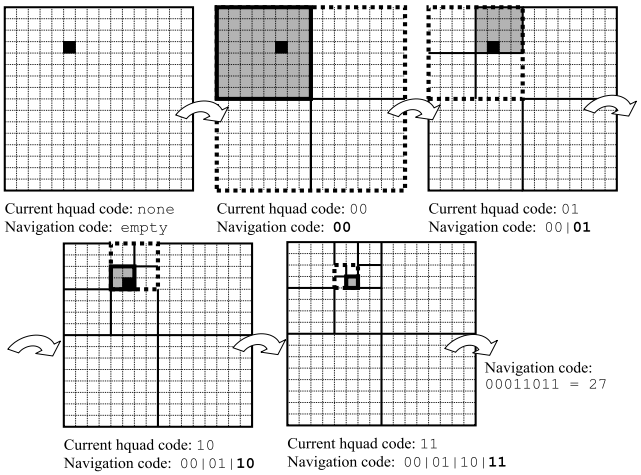

Figure 9. Navigation code construction for point hquad located at coordinates [5,3].

information (e.g. in image processing or pattern recognition).

3.1.3 Important Consequences

Let us summarize important consequences that will be important in further analysis. The consequences presented can be also observed from Figure 10.

- The hquads define in the space Ω Z-regions which accurately "fit" the hquads.
- The hquads on a given HQ-tree level are Z-ordered relative to their navigation codes.
- Let us consider an hquad on a k-th level of an HQ-tree. The Z-curve will visit its sub-hquads on $(k+1)$-th level one-by-one, i.e. the hquad is entirely "filled" before the Z-curve will enter the next hquad on the k-th level.

3.2 Minimal Z-region Hquad Envelope

Now we can describe an algorithm for testing the query box and a Z-region intersection. This algorithm follows the presented geometric model and is based on a construction of *minimal Z-region hquad envelope*.

Definition 7
Every Z-region can be assembled by some number of hquads from various levels of the HQ-tree. A set of hquads forming given Z-region we call *Z-region hquad envelope*. Envelope formed by the smallest number of hquads we call *minimal Z-region hquad envelope*.

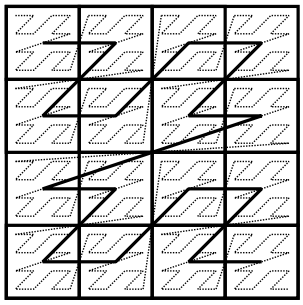

Figure 10. Z-ordered hquads of the first and third level of an HQ-tree.

First, we briefly sketch the algorithm for creating a minimal Z-region hquad envelope (see also Figure 11).

Phase 1. Find the smallest hquad hq_{min} common to both Z-region bounds α and β.
Phase 2. Send all the sub-hquads of hq_{min} lying between the Z-region bounds α, β to the output.
Phase 3. Process separately the Z-region bounds α and β. The sub-hquad hq of hq_{min} containing the respective bound is processed using the HQ-tree downward pass by one level in each step. The hquad hq is divided on two groups of its sub-hquads. A sub-hquad of hq which contains the bound is denoted as "dividing hquad" hq_{div}. The *lower group* is formed by sub-hquads of hq the hquad codes of which are lower than hquad code of hq_{div}. Similarly, the *upper group* is formed by sub-hquads with codes greater than code of hq_{div}. In the case of α (β respectively) processing, the upper (lower respectively) group is sent to the output in each step. After the step, hq is set to hq_{div}. The steps are repeated until hq becomes a point.

In the presented description we have assumed that every minimal Z-region envelope must contain at least two point hquads. There also exist Z-regions whose minimal envelopes consist of only hquads located on the higher levels of the HQ-tree, e.g. the envelope for the whole space Ω consist of single hquad. Such "simple" Z-regions can be handled in the second and third phase of the algorithm but for the sake of simplicity we can omit these cases.

Second, we present a more detailed algorithm.

Algorithm 2 (minimal hquad envelope construction)

```
CreateEnvelope(Hquad space, Zaddress alpha, Zaddress beta)
{
  Hquad topHquad = space
  i = 0
  /* phase 1 */
  While (alpha[i] = beta[i]) {
    topHquad = topHquad.GetSubHquad(alpha[i])
    i++
  }
```

```
  /* phase 2 */
  // send all hquads between α and β
  For j=alpha[i]+1 To beta[i]-1
    AddToEnvelope(topHquad.GetSubHquad(j))

  /* phase 3 */
  Hquad topAlphaHquad = topHquad.GetSubHquad(alpha[i])
  Hquad topBetaHquad = topHquad.GetSubHquad(beta[i])
  For j=i+1 To log2(card(D))-1 { // HQ-tree pass
    // send all hquads after α
    For k=alpha[j]+1 To 2^n-1
      AddToEnvelope(topAlphaHquad.GetSubHquad(k))
    topAlphaHquad = topAlphaHquad.GetSubHquad(alpha[j])
    // send all hquads before β
    For k=0 To beta[j]-1
      AddToEnvelope(topBetaHquad.GetSubHquad(k))
    topBetaHquad = topBetaHquad.GetSubHquad(beta[j])
  }
  // send the point hquads of α and β
  AddToEnvelope(topAlphaHquad)
  AddToEnvelope(topBetaHquad)
}
```

Notes to the pseudo-code:

- `alpha[i]`, `beta[i]` returns the i-th hquad code in the Z-address.
- `topHquad.GetSubHquad(i)` returns sub-hquad of `topHquad` identified by hquad code i.

Using the Algorithm 2 an envelope is created. If we replace the operation `AddToEnvelope` with operation `TestQueryBoxInHquad`, we will get the desired function testing the intersection between a Z-region and the query box.

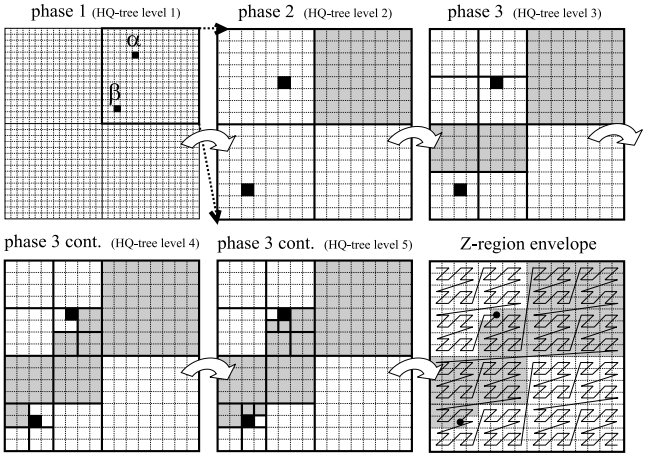

Figure 11. Construction of the minimal Z-region hquad envelope. The Z-region $[\alpha : \beta]$ is formed by 13 hquads. The Z-region bounds α, β determine the "dividing hquads" for each step (HQ-tree level respectively) in the third phase.

From another point of view, we can imagine the envelope as a subtree in the HQ-tree. Figure 12 presents such a subtree to the above mentioned example. We can see that in the 3rd phase (levels 2-4) the left branch for the α bound and right branch for the β bound are passed down. In the left branch the upper groups of sub-hquads are sent to the output (the grayed parts) while in the right branch the lower groups of sub-hquads are grayed and thus sent to the output.

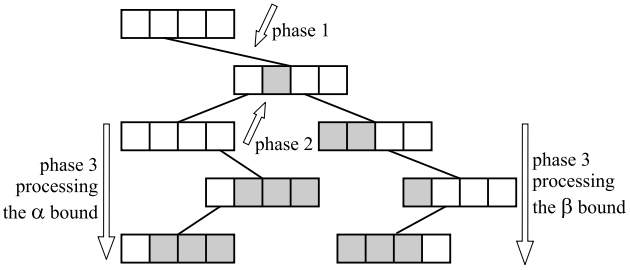

Figure 12. The hquad envelope subtree.

3.2.1 Intersection of Hyper-Boxes

The operation `TestQueryBoxInHquad` is used for testing whether the query box is intersecting a given hquad. This test is evaluated as an intersection of two hyper-boxes.

As we can see in Figure 13a, two hyper-boxes are intersecting just in case that their ranges intersect in *all* dimensions. For the ranges of a particular dimension three states may occur, see Figure 13b. If we denote the first hyper-

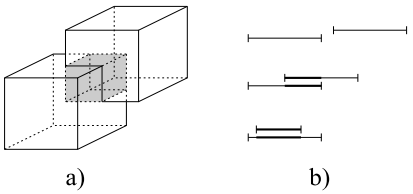

Figure 13. Intersection of two hyper-boxes.

box range of the i-th dimension as an interval $\langle low_1^i, up_1^i \rangle$ and range of the other hyper-box as an interval $\langle low_2^i, up_2^i \rangle$ we can formulate a single condition when the two ranges are intersecting:

$$|low_1^i - low_2^i| + |up_1^i - up_2^i| \leq |low_1^i - up_1^i| + |low_2^i - up_2^i|$$

Furthermore, the two hyper-boxes are intersecting iff the statement holds for all i.

3.2.2 Time Complexity

The first two phases of the ”envelope algorithm” we can omit since their complexities are lower than complexity of the third phase.

In the third phase there are four nested loops. The first one is passing down the HQ-tree, i.e. the height of the tree is $h = log_2(card(D))$. The second one is sending up to $2^n - 1$ sub-hquads to the output in each level. Finally, the third and fourth loop is hidden in the intersection test of two hyper-boxes (testing n coordinates each consisting of $log_2(card(D))$ bits) – i.e. $O(n * log(card(D)))$.

The overall time complexity is a product of the particular nested complexities:

$$O(n * 2^n * log^2(card(D)))$$

This complexity would be acceptable if we use vector space with small dimensionality, e.g. $n < 10$. However, for high-dimensional spaces this algorithm is not suitable. Fortunately, there exists a solution reducing the complexity to linear according to the Z-address bit length as we will see in the next section.

3.3 Linear Intersection Algorithm

The greatest component of the "envelope algorithm" complexity is the exponential dependence on n. This fact arises from the observation that the *lower* or the *upper* group sent to the output can consist of up to $2^n - 1$ hquads.

However, we have found out that the lower (upper respectively) group can be spatially represented with at maximum n general hyper-boxes (not hquads yet). The idea is based on successive elimination of all dimensions while processing the hquad code of the "dividing hquad".

We will present the idea for the upper group construction. The lower group is constructed dually.

1. Active hyper-box is set to the parent hquad.
2. Bits are iteratively read from the hquad code of the "dividing sub-hquad" from the most significant to the less significant. We already know that each bit of the code determines location of the hquad relative to appropriate dimension.

   In each step, we process next bit of the hquad code (the steps are repeated until all bits, i.e. dimensions, are processed). For each bit value, two cases can occur:

   (a) If the bit is set to `0` it means that the "dividing hquad" is located in the first (lower) half of appropriate dimension. Since we construct the upper group we can send to the output the upper (relative to the dimension) half of the active hyper-box. The active hyper-box is then set to its own lower half.

   (b) If the bit is set to `1` it means that the "dividing hquad" is located in the second (upper) half of appropriate dimension. Since we construct the upper group we can ignore the lower half and the active hyper-box is set to its own upper half.

The algorithm will construct maximally n hyper-boxes for the upper group. In consequence, constructed hyper-box can hold space consisting of up to 2^{n-1} hquads, i.e. a half of space. See examples in Figure 14.

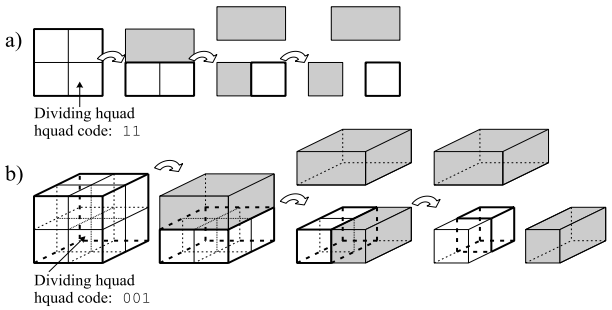

Figure 14. a) 2D example of lower group construction. b) 3D example of upper group construction.

We will use this particular lower/upper group constructions for the linear intersection algorithm.

Phase 1. The first phase is the same as by the exponential algorithm. Smallest hquad entirely containing the Z-region is determined. This hquad is bounded with some interval $[\alpha_{hq} : \beta_{hq}]$ and further process is restricted into this hquad.
Phase 2. The second phase will construct so called *lower* and *upper* Z-region *half-envelopes*. Lower half-envelope is created for the upper bound β and upper half-envelope is created for the lower bound α. This half-envelopes are constructed using the above described upper/lower group construction on each level of the HQ-tree. The algorithm ensures that both half-envelopes are disjunct.

Geometric union of the two half-envelopes gives a "hyper-box envelope" spatially matching the "hquad envelope". This fact also states that if the query box intersect the Z-region then it must also intersect at least one of the the half-envelopes and vice versa.

Algorithm 3 (linear intersection algorithm)

```
bool TestZRegionIntersection(Zaddress alpha, Zaddress beta,
Hbox spaceBox, Hbox queryBox)
{
  /* phase 1 - reduce the spaceBox to the smallest
  hquad entirely containing the Z-region */
     ...(see phase 1 in Algorithm 3)...

  /* phase 2 - intersection of half-envelopes */
  If (TestUpperHalfEnvelope(spaceBox, queryBox, alpha) Or
      TestLowerHalfEnvelope(spaceBox, queryBox, beta)) Then
         return true
  Else
         return false
}
```

```
bool TestUpperHalfEnvelope(Hbox activeHBox,
Hbox queryBox, Zaddress alpha, Zaddress beta)
{
  // HQ-tree downward pass
  For i=0 To log2(card(D))-1 {
    For j=n-1 To 0 { // process all dimensions
      If (alpha[i].GetBit(j)) Then {
        // set the activeHBox to its upper half
        activeHBox = activeHBox.GetUpperHalf(j)
      }
      Else {
       // test the intersection with the query box
        If (flag And TestQueryBox(activeHBox.GetUpperHalf(j),
              queryBox)) Then
          return true
        If (TupleInsideBox(beta, activeHBox.GetUpperHalf(j))
          flag = true
        activeHBox = activeHBox.GetLowerHalf(j)
      }
    }
  }
  // test the last point (alpha)
  return TestQueryBox(activeHBox, queryBox)
}
```

The linear algorithm is depicted in Figure 15.

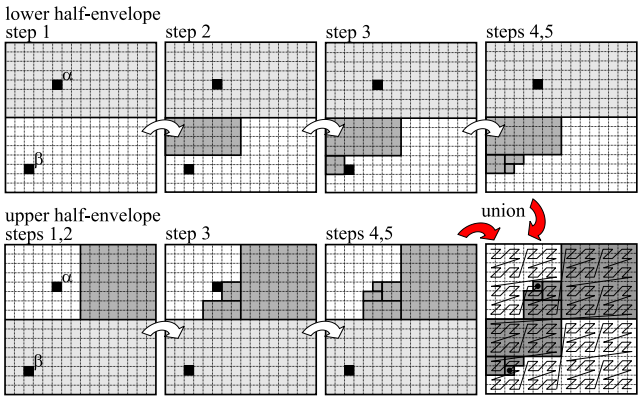

Figure 15. The linear intersection algorithm. Resultant envelope consists of 9 hyper-boxes, i.e. max. 9 hyper-boxes are tested for an intersection.

Note that hyper-boxes are sent to the output after detection that active hyper-box cannot contain the opposite Z-region bound. Until then, boxes are ignored (see the light-gray boxes in the Figure 15). This restriction (in the pseudo-code provided using the `flag` variable) ensures that both constructed envelopes are disjunct. It can be proved that the "hyper-box envelope" constructed in such way is spatially identical to the "hquad envelope".

3.3.1 Time Complexity

As we have said earlier the complexity was reduced. It is now $O(n*log(card(D)))$. There are only two nested loops while the first one is the HQ-tree pass and the second one is testing all dimensions of a "dividing hquad" code.

Note that the operations `TestQueryBox` and `TupleInsideBox` can be realized in $O(1)$ time because only *one* dimension is checked in each step and the range of the active hyper-box is always *halved*.

Had we relate the complexity to the Z-address bit length, i.e. to $|Zaddr| = n * log_2(card(D))$, it would be linear.

4 Experimental Results

Before we present the experiments[2], we must specify necessary details about the DRU algorithm – an optimized version of the algorithm sketched in Section 2.

4.1 The 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 simply "jumps right" (leafs are linked) to the neighbour leaf and continues processing. This optimization was already used in the Bayer-Markl's algorithm.

The second optimization, called *neighbour region*, is specific to the DRU algorithm (to the `TestZregionIntersection` operation respectively). It 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. It should be noticed that the *neighbour region* optimization is applicable also on the original Bayer-Markl's algorithm, but this fact has never been published or even mentioned in previous works and thus we do not deal with such an optimized version of Bayer-Markl's algorithm.

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. 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.

[2]Comparison of the UB-tree with other multidimensional structures was out of scope of this paper, we refer to [1, 10] where the superiority of UB-tree (over R-tree, Grid-files, etc.) was already presented.

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 (using halving the interval) 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}).

4.2 Testing a Synthetic Dataset

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.

UB-tree characteristics (synthetic datasets):

$card(D)$	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

Query boxes of various shapes were generated randomly according to the distribution of tuples in space. Ranges of query boxes were fixed for growing dimensionality thus the volumes were increasing but the query box volume/space volume ratio 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.

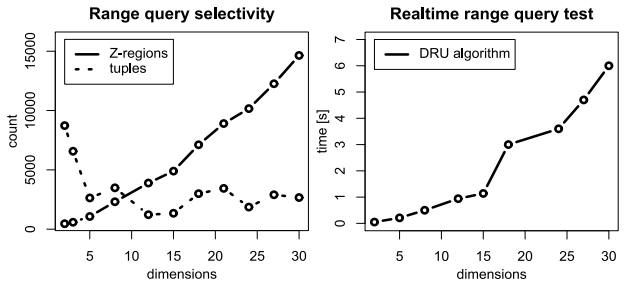

Figure 16. a) Range query selectivity. b) Realtime range query test.

In Figure 16a, the range query selectivity is presented. Note that the number of returned tuples is approximately constant but the number of accessed Z-regions (retrieved leafs respectively) rapidly grows with increasing dimensionality. The Figure 16b shows real times of the range query execution[3]. In Figure 17 the original Bayer-Markl's

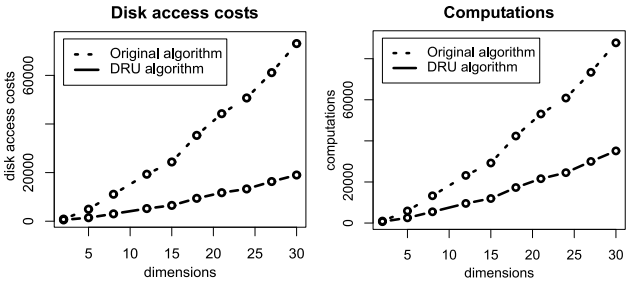

Figure 17. a) Disk access costs. b) Computations.

algorithm and the DRU algorithm are compared. Figure 17a shows number of disk accesses and Figure 17b shows the number of computations.

The results demonstrate that the DRU algorithm performs better since the DAC and computation growth is much less steep than by the Bayer-Markl's algorithm. The reason of that behaviour can be observed from Figure 18 where the force of leaf optimizations is presented.

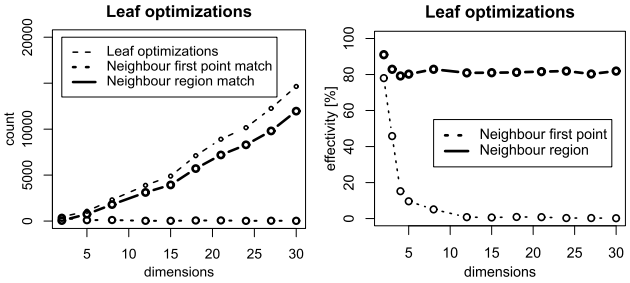

Figure 18. a) Leaf optimizations. b) Leaf optimizations effectivity.

Figure 18a shows how many times the optimizations took an effect. For dimensionality $n = 2$, the *neighbour first point* optimization detected the majority of the relevant neighbour regions. However, for $n > 4$ the *neighbour first point* optimization became almost useless. This was expectable since the shape of a Z-region is much more complicated for the higher dimensionalities.

On the other side, the "stronger" *neighbour region* optimization works for any dimensionality and thus saves many unnecessary operations as well as disk accesses. Figure 18b shows the effectivity of both optimizations. The effectivity

[3]Measured on Intel Pentium®4 2.4Ghz, 512MB DDR333, WinXP

of the *neighbour region* optimization means that cca 80% of tested neighbour Z-regions were intersecting the query box. However, the effectivity of the *neighbour first point* optimization says that for $n > 10$ the first points of the neighbour regions are always outside the query box.

Presented results allow us to think about the *curse of dimensionality* [2, 14] appearing in the UB-tree. With the growing dimensionality of UB-trees grow also the costs, even though less than exponentially. Figure 19a presents a ratio of tuples inside the query box to the number of intersecting Z-regions. Figure 19b 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. On the other side, the Figure 19b

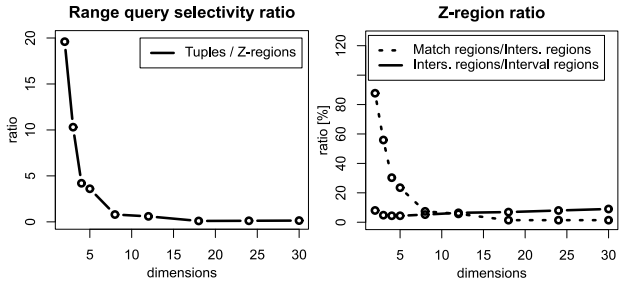

Figure 19. a) Range query selectivity ratio. b) Z-region ratio.

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* [7] used in many applications is very affected by the curse of dimensionality and its usage for high-dimensional indexing is nearly impossible.

5 Conclusions And Outlook

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. For the DRU algorithm, we have introduced the Z-region intersection algorithm testing whether the Z-region is intersecting a given query box.

In the future, we are going to develop additional algorithms for the UB-trees based on the Z-region intersection algorithm. Furthermore, we would like to modify the Z-region intersection algorithm for other space filling curves with better clustering properties than the Z-curve.

References

[1] Bayer R. The Universal B-Tree for multidimensional indexing: General Concepts. In: *Proc. Of World-Wide Computing and its Applications 97 (WWCA 97)*. Tsukuba, Japan, LNCS 1274, Springer-Verlag, 1997.

[2] Böhm C., Berchtold S., Keim D. Searching in High-Dimensional Spaces – Index Structures for Improving the Performance of Multimedia Databases. In: *ACM Computing Surveys (33)3*, pp. 322–373, 2001.

[3] Faloutsos C., Roseman S. Fractals for secondary key retrieval. In: *Proc. 8th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pp. 247-252, 1989.

[4] Fenk R. The BUB-Tree. In: *Poster at the 28th Conference VLDB*, Hongkong, 2002

[5] Gaede V., Günther O. Multidimensional Access Methods, In: *ACM Computing Surveys (30)2*, 1998.

[6] Gotsman C., Lindenbaum M. On the Metric Properties of Discrete Space-Filling Curves, In: *Proc. of IEEE Int. Conference on Pattern Recognition*, Vol III, pp. 98–102, Jerusalem, Israel, 1994.

[7] Guttman A. R-Trees: A Dynamic Index Structure for Spatial Searching, In: *Proc. of ACM SIGMOD 1984, Annual Meeting*, pp. 47–57, Boston, 1984.

[8] Lawder J., King P. Querying Multi-dimensional Data Indexed Using the Hilbert Space-filling Curve. *SIGMOD Record 30(1)*, 2001.

[9] Lawder J., King P. Using Space-Filling Curves for Multi-dimensional Indexing, In: *Proc. of BNCOD 2000*.

[10] Markl V. *Mistral: Processing Relational Queries using a Multidimensional Access Technique*, thesis, 1999.

[11] Ramsak F., Markl V., Fenk R., Zirkel M., Elhardt K., Bayer R. Integrating the UB-tree into a Database System Kernel. In: *Proc. Of the 26th Int. Conference VLDB*, Cairo, Egypt, pp. 263–272, 2000.

[12] Sagan H. *Space-Filling Curves*, Springer, 1994.

[13] Samet H. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[14] Yu C. *High-Dimensional Indexing*, Springer–Verlag, LNCS 2341, pp. 9–33, 2002