Available online at www.sciencedirect.com
SCIENCE DIRECT®
Information Systems 31 (2006) 489–511
www.elsevier.com/locate/infosys

A new range query algorithm for Universal B-trees ☆

Tomáš Skopal[a,*], Michal Krátký[b], Jaroslav Pokorný[a], Václav Snášel[b]

[a]*Department of Software Engineering, Charles University in Prague, Malostranské nám. 25, 118 00 Prague, Czech Republic*
[b]*VŠB—Technical University of Ostrava, Department of Computer Science, 17. listopadu 15, 708 33 Ostrava, Czech Republic*

Received 1 March 2004; received in revised form 12 November 2004; accepted 8 December 2004

Abstract

In multi-dimensional databases the essential tool for accessing data is the range query (or window query). In this paper we introduce a new algorithm of processing range query in universal B-tree (UB-tree), which is an index structure for searching in multi-dimensional databases. The new range query algorithm (called the DRU algorithm) works efficiently, even for processing high-dimensional databases. In particular, using the DRU algorithm many of the UB-tree inner nodes need not to be accessed. We explain the DRU algorithm using a simple geometric model, providing a clear insight into the problem. More specifically, the model exploits an interesting relation between the Z-curve and generalized quad-trees. We also present experimental results for the DRU algorithm implementation.

Keywords: Spatial access methods; Multi-dimensional indexing; Range query; UB-tree; DRU algorithm; Z-region; Space filling curves

1. Introduction

In the area of database systems, the emergence of new database forms requires a development of appropriate access methods. Unlike single-dimensional databases, which are indexed/searched according to a simple key (e.g. using B-trees), we often intend to access data according to a composite key (according to several attributes generally). We call such databases *multi-dimensional*, since data instance is represented by a vector of simple values. A collection of data vectors (data tuples) can be interpreted as set of points in a multi-dimensional vector space.

1.1. Preliminaries

Let us specify several necessary notations needed for further discussion:

Definition 1 (*vector space*). A *discrete vector space* Ω is defined as the cartesian product of finite domains D_i, i.e. $\Omega = D_1 \times D_2 \times \cdots \times D_n$. The vector space Ω has n dimensions, while each

☆ Recommended by Patrick O'Neil, Area Editor.

*Corresponding author. Fax: +420 221 914 323.
E-mail addresses: tomas@skopal.net (T. Skopal), michal.kratky@vsb.cz (M. Krátký), jaroslav.pokorny@mff.cuni.cz (J. Pokorný), vaclav.snasel@vsb.cz (V. Snášel).

0306-4379/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2004.12.001

particular domain D_i is associated with the ith dimension of the space. Each *point* (in the universe Ω) or *data tuple* (in the database) is represented by a vector $o = [o_1, o_2, \ldots, o_n]$, $o_i \in D_i$.

For the sake of simplicity, we assume that the vector space Ω is a hyper-cube determined as the nth power of a single domain D, i.e. $\Omega = D^n$, where D is a linearly ordered interval of integers $D = \langle 0, 2^p - 1\rangle$. The cardinality of D is $|D| = 2^p$ for some integer p.

1.2. Range query

One of the most popular queries required for access to multi-dimensional databases is the *range query* (also called window query or rectangular query), by which the user specifies an interval of values $\langle a_i, b_i\rangle$ (for each attribute A_i), which the retrieved data tuples have to match. The range query can be represented by a hyper-box QB in the space Ω. The ranges of query box QB are defined by two boundary points, the lower bound $\mathrm{QB}_{\mathrm{low}} = [a_1, a_2, \ldots, a_n]$ and the upper bound $\mathrm{QB}_{\mathrm{up}} = [b_1, b_2, \ldots, b_n]$, where $a_1 \leqslant b_1, a_2 \leqslant b_2, \ldots, a_n \leqslant b_n$. The purpose of range query is to select all data tuples inside the query box QB, i.e. to select all such tuples o satisfying $a_i \leqslant o_i \leqslant b_i$, for $1 \leqslant i \leqslant n$ (see Fig. 1).

However, for range queries the classic indexing methods, maintaining n single-dimensional indices, are inefficient. For that reason, a class of indexing methods has been developed, called *spatial access methods*[1] (SAMs), allowing to efficiently index and query multi-dimensional data.

In this paper we introduce a new range query algorithm for the universal B-tree (UB-tree), which is a spatial access method based on the B$^+$-tree and the Z-ordering. With the new algorithm (called the DRU algorithm), we address two issues. First, the existing algorithms are either inefficient or vaguely described. Our approach, on the other side, is deeply described in the geometric as well as in the algorithmic way. Second, the DRU algorithm works more efficiently in high-dimensional indices.

[1]For a general survey over various SAMs we refer to Gaede and Günther [1] or to Böhm et al. [2].

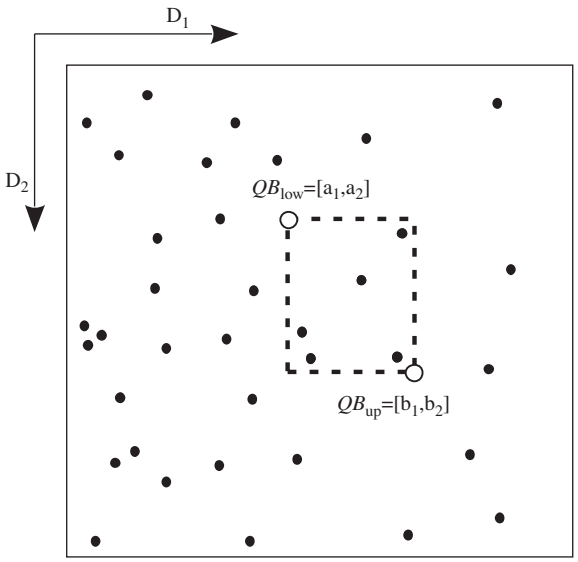


Fig. 1. 2D query box QB specified with lower bound $\mathrm{QB}_{\mathrm{low}}$ and upper bound $\mathrm{QB}_{\mathrm{up}}$.

The paper is organized as follows. In Section 2 we overview the basic concepts of UB-tree. The problem of range query processing and some related work is discussed in Section 3. The description of the DRU algorithm is presented in Section 4, while the geometric principles behind the algorithm are explained in Section 5. In Sections 6, 7 the experimental results are analysed and concluded.

2. UB-tree

In Bayer [4], the Universal B-tree[2] has been introduced as a data structure for indexing multi-dimensional databases. In simple words, the UB-tree can be characterized as a combination of the well-known B$^+$-tree with the Z-ordering. Using the Z-ordering, each multi-dimensional data tuple is transformed into an integer (called Z-address), which is inserted into the B$^+$-tree. That is, the problem of searching in multi-dimensional

[2]A modification called *bounding* UB-*tree* (BUB-tree) has been recently introduced [3], allowing to avoid indexing of the "dead space" (uselessly indexed empty space).

database is turned into the problem of searching in ordered set (indexed by the B^+-tree).

2.1. Motivation

The ideas of indexing multi-dimensional databases as single-dimensional databases have been appeared long ago [5–8], while the motivation was to reuse the power of existing linear index structures (like B-tree, AVL-tree, or simple sorted array) in order to answer range queries (or spatial queries, in general). To provide this functionality, each multi-dimensional data tuple is transformed (possibly in a reversible way) into a single integer and inserted into the respective linear index structure. The transformation is provided using a discrete *space filling curve* (SFC) [9], which defines a linear ordering of all the points of the multi-dimensional vector space. In other words, to each point in the space an *address* is assigned, defining a global order of the point in space. This address is a unique number defining position of the point on curve. In case of Z-ordering (Z-curve, respectively), the address of a point is called *Z-address*.

2.2. Z-regions

In addition to the simple combination of the B^+-tree and the Z-ordering, the concept of UB-tree gives an interpretation to the nodes of B^+-tree. Each node represents, in fact, an interval $[\alpha : \beta]$ (α is the lower bound, β is the upper bound) on the Z-curve (in the Z-ordering, respectively), called the *Z-region*. Since Z-address represents a point in the space, Z-region represents an area in the space. Finally, the inner nodes of UB-tree recursively partition the space, such that we obtain a hierarchy of nested Z-regions. The balanced hierarchy of Z-regions is similar to some other spatial structures (e.g. the R-tree), however, a particular advantage of UB-tree is that Z-regions are formed implicitly by the B^+-tree behaviour, i.e. no heuristics is needed as in case of splitting R-tree's minimum bounding rectangles. Another property is that all Z-regions at a given UB-tree level do not overlap. An example of the two-dimensional Z-curve and several Z-regions is presented in Fig. 2a (the numbers represent Z-addresses).

Each Z-region in the UB-tree is mapped into a single node (disk page) in the underlying B^+-tree hierarchy. The UB-tree leaves represent the Z-regions containing the indexed data tuples themselves, while the inner nodes represent the super-Z-regions. A *super-Z-region* spatially bounds all the (super-)Z-regions present in the appropriate subtree. An example of UB-tree for the collection of data tuples from Fig. 2a is depicted in Fig. 2b.

2.3. Evaluation of range query in UB-tree

The implementation of basic operations on UB-tree (i.e. insertion, deletion, point query) is analogous to the implementation of operations on the "ordinary" B^+-tree. The main difference is that in UB-tree we must first compute Z-address of

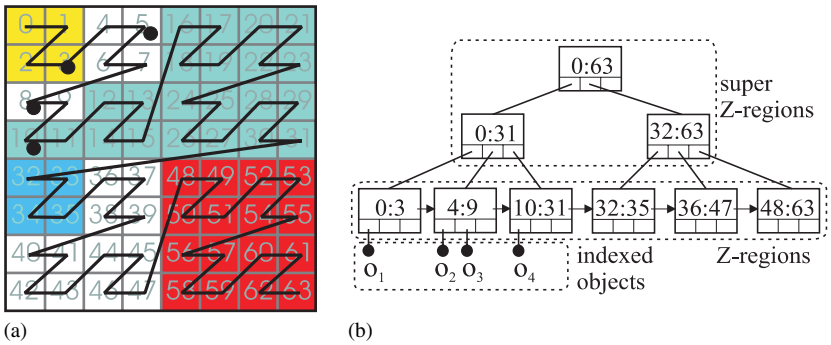


Fig. 2. (a) Two-dimensional space 8×8 filled by the Z-curve, partitioned into six Z-regions. (b) The UB-tree nodes correspond to the Z-regions and super-Z-regions.

the inserted/deleted/queried tuple, in order to obtain a simple key for the subsequent operation on the underlying B^+-tree.

On the other side, a range query cannot be so simply forwarded to the B^+-tree. This arises from the exclusivity of range query, which is intended to be used on multi-dimensional data structures only.

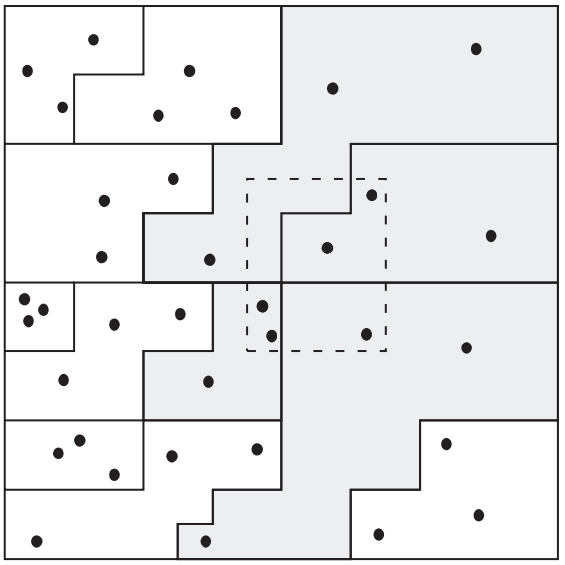


Fig. 3. Space Ω partitioned into Z-regions. The query box intersects four Z-regions.

In the context of UB-trees, the range query evaluation can be reformulated as a search over all such leaf pages, the Z-regions of which intersect the query box QB (see Fig. 3).

3. Related work

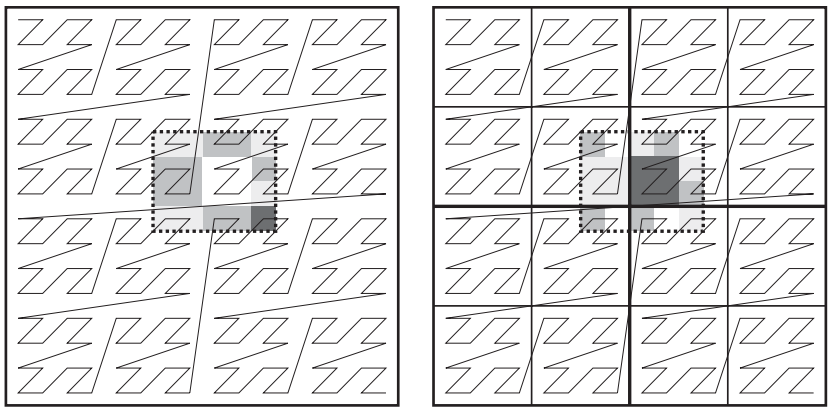
Although there has been already proposed a range query algorithm together with the introduction of UB-tree, in the following subsection we begin the discussion with an earlier and more general approach.

3.1. Query box decomposition

According to several early works, the problem of range query evaluation can be turned into the problem of decomposition of the query box into a set of virtual Z-regions, which exactly match the space of the query box. Using the set of Z-regions, the range query is evaluated either as a complex interval query, or as many simple interval queries.

In Orenstein and Merrett [5] the authors propose a recursive decomposition algorithm, in which a minimal set of rectangular Z-regions is formed, spatially matching the query box (see an example in Fig. 4a).

In another approach (as referred e.g. in [10]) the query box is decomposed into a subtree of multi-dimensional quad-tree. Since each (sub)quadrant



(a) (b)

Fig. 4. Query box decomposition: (a) rectangular Z-regions; (b) quadrants.

in the quad-tree is a special case of Z-region,[3] the subtree represents a set of Z-regions matching the query box (see Fig. 4b).

The idea of query box decomposition is general in sense, that it is data-independent and structure-independent. In other words, the query box is decomposed regardless of the database being queried. Moreover, it does not matter which index structure is chosen for the query evaluation; the only requirement is that the structure must be able to maintain linear order of keys.

On the other side, there is a serious disadvantage of the query decomposition approach; the number of Z-regions forming the query box is exponentially dependent on the dimensionality which, in turn, means exponential time complexity of the query evaluation. This property is not so critical for small dimensionalities (e.g. up to 5), however, for high-dimensional databases this becomes a serious limitation. In Orenstein and Merrett [5] the total costs of query evaluation have been heuristically reduced, nevertheless, the time complexity remained exponential.

3.2. Query box touching

In this subsection we present a more detailed description of the original UB-tree range query algorithm (denoted as Bayer–Markl's algorithm), which has been introduced in Bayer [4] and improved in Markl [11]. The algorithm is data-dependent and also structure-dependent—it sequentially examines all UB-tree leaves, the Z-regions of which intersect the query box. The query-intersected leaves are consecutively retrieved by a sequence of point queries. Each point query is specified by the smallest Z-address lying inside the query box and being greater than β of the previously processed Z-region. Since the point queries "touch sides" of the query box, we call the approach "query box touching".

Algorithm 1 (*Bayer–Markl's range query algorithm*).

Input: UB-tree, query box QB

Output: data tuples inside QB

(1) Z-address of the query box lower bound is computed, i.e. $Z_{\text{val}} = Z_{\text{addr}}(\text{QB}_{\text{low}})$.
(2) The following steps are repeated as long as the Z_{val} is lower or equal than Z-address of the query box upper bound, i.e. while $Z_{\text{val}} \leqslant Z_{\text{addr}}(\text{QB}_{\text{up}})$
   (a) At the deepest UB-tree level a page P is retrieved, the Z-region of which contains Z_{val}, i.e. $\alpha \leqslant Z_{\text{val}} \leqslant \beta$.
   (b) Page P is searched for all data tuples lying inside QB. These tuples go to the output as a part of the result.
   (c) The next intersected Z-region must be determined. The smallest Z-address greater than β and intersecting the query box must be found—an operation `GetNextZaddress`(β, QB) is performed—and the result is stored in Z_{val}.

In Fig. 5, a running of the Bayer–Markl's algorithm is shown. At first, Z-address of the query box lower bound is computed. Using this value, a leaf page of the UB-tree is retrieved and searched for relevant data tuples. Next, the following query-intersected leaf is retrieved and so on. The algorithm will finish as soon as the β of the actual Z-region gets greater than Z-address of the query box upper bound, i.e. when $\beta > Z_{\text{addr}}(\text{QB}_{\text{up}})$.

So far, the algorithm description was quite clear. A problem arises if we look deeper into the operation `GetNextZaddress`. The computation of the smallest Z-address inside the query box is not trivial, since this procedure is obviously dependent on the shape of Z-region. The algorithm for `GetNextZaddress` proposed by Bayer [4] is of exponential time complexity (according to the dimensionality). Later [11,12], the authors presented a version that is of linear time complexity, according to the Z-address bit-length (i.e. to $|Z_{\text{addr}}| = n \log_2(|D|)$).

Unfortunately, all descriptions of the linear `GetNextZaddress` published so far have been mentioned very briefly and can be hardly used as a guide for real implementation of the `GetNextZaddress` algorithm. Moreover, the explanations

[3] This property is reflected also in our approach, as we explain in Section 5.

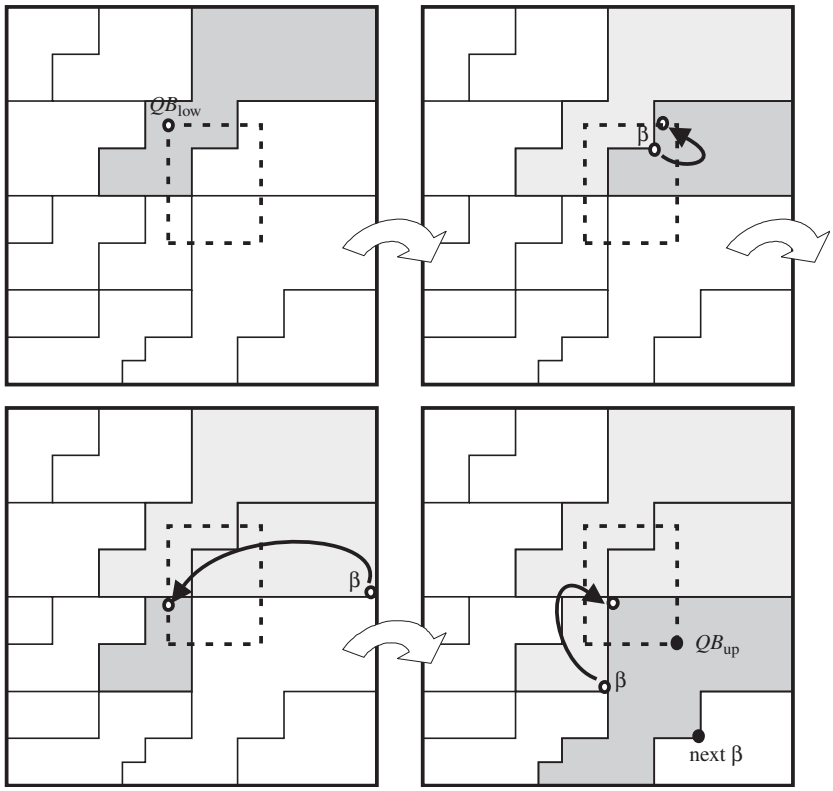


Fig. 5. Processing of the Bayer–Markl's range query algorithm. The β bounds of the consecutively retrieved intersected Z-regions are computed as long as $QB_{low} \leqslant \beta \leqslant QB_{up}$.

have always been based on a pure algorithmic basis using "handling with bits", hence lacking a geometric model providing a deeper abstract view. The unclearness about the linear `GetNextZaddress` algorithm is perhaps intentional, since original algorithms on UB-trees are protected by international patents.[4]

3.2.1. Time complexity

Let h be the height of UB-tree, m be the number of data tuples stored in UB-tree leaves, and $c \geqslant 2$ be a fixed node capacity (arity of the UB-tree), i.e. $\log_c(m) - 1 \leqslant h \leqslant \log_2(m) - 1$. Let k be the number of Z-regions intersected by the query box. The time complexity of the `GetNextZaddress` operation is linear according to the Z-address bit-length, i.e. it is $O(n \log(|D|))$. In each step, the algorithm retrieves the next of the k intersected Z-regions. This operation consists of one calculation of the `GetNextZaddress` and of one UB-tree downward traversal (point query respectively) required for the query-intersected leaf retrieval. Thus, the overall time complexity of the range query is $O(k \cdot h \cdot n \log(c) \log(|D|))$.

3.2.2. I/O costs

The number of I/O operations spent by any database-oriented algorithm has an important impact on the overall efficiency. In case of the range query algorithm, the I/O operation is represented as a single disk page retrieval. During the presented algorithm running, each query-intersected leaf is retrieved by h disk page retrievals. The leaf search is performed k times, thus the overall I/O costs are $I/Os = k \cdot h$.

[4]Deutsches Patentamt Nos. 197 09 041.9 and 196 35 429.3.

3.2.3. Other related work

Almost the same algorithm as the Bayer–Markl's one was presented in Lawder and King [13,14], but the authors have not used the UB-tree as a framework. They have applied two types of space filling curves to their 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, because also these works explain it very vaguely.

4. The down-right-up algorithm

In our approach, we have focused on the basic straightforward idea that range query must search only such leaves, the Z-regions of which intersect the query box. This can be performed via a single UB-tree downward traversal. The UB-tree is traversed 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. Only the intersected nodes are further processed. At the leaf level, all data tuples located inside the query box are returned as the query result. The idea is outlined in Fig. 6.

Like the original algorithm, also our idea is conditioned by a specific crucial operation. This particular operation (denoted as `TestZRegionIntersection`) examines, whether a given Z-region does intersect the query box or does not. We analyze this operation closely in Section 5.

More specifically, the algorithm, called the *Down-Right-Up* (*DRU*) *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 α of the right-neighbour-leaf's Z-region lies inside the query box. If it does, the algorithm simply "jumps right" (the leaves are linked) to the neighbour leaf and the processing goes on. This kind of optimization was already utilized by 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 right-neighbour-leaf's Z-region is intersected by the query box. If it is, the algorithm "jumps right" similarly like by the first optimization. It should be

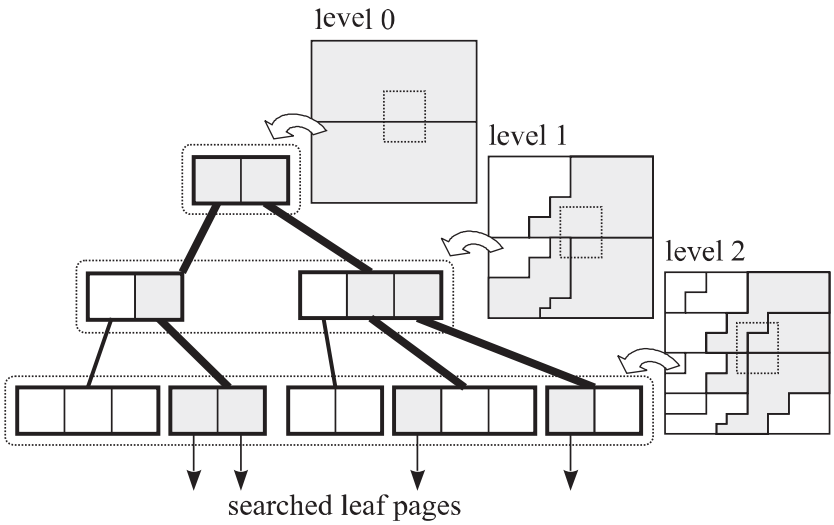


Fig. 6. Single-pass range query processing. Only the intersected Z-regions (nodes respectively) are processed. These Z-regions (nodes) are greyed. The bold branches are the only paths traversed down.

remarked, that an optimization similar to the *neighbour region* could be possibly utilized, in principle, also by the original Bayer–Markl's algorithm, but this fact has never been mentioned in previous works, so we do not cope with such an optimized version of the original algorithm.

Algorithm 2 (*The DRU algorithm*). In order to keep the actual UB-tree path being processed, the algorithm uses the *path stack*. The path stack allows us to avoid accesses to the nodes (and to the entries in nodes) already processed.

Input: UB-tree, query box QB
Output: data tuples inside QB

(1) Find a leaf the Z-region of which contains $Z_{\text{addr}}(\text{QB}_{\text{low}})$. Store the path on path stack and set the retrieved leaf as the actual leaf.
(2) Search the actual leaf for data tuples lying inside QB[5] and return the tuples as a part of the result.
(3) If α of the right-neighbour-leaf's Z-region lies inside QB, then retrieve the right neighbour leaf, set it as the actual leaf and goto step 2. This is the *neighbour first point* optimization.
(4) If the right-neighbour-leaf's Z-region intersects QB, then retrieve the right neighbour leaf, set it as the actual leaf and goto step 2. This is the *neighbour region* optimization.
(5) The stack must be recovered after the "optimization jumps". The UB-tree is traversed to the next query-intersected node (along the path on the stack). After the recovery, the top of stack contains parent node of the last leaf reached by the preceding optimization(s).
(6) Peek node N on the top of stack and try to find an entry in N, pointing to the next query-intersected node R. The entry is tried to find (using halving the interval) as the first right-hand entry, the Z-region of which intersects QB. If no such entry is found, remove node N from the stack and repeat step 6 (the **Up-Phase**). If such an entry is found, retrieve the node R (the **Right-Phase**) and push it onto the stack (the **Down-Phase**). If R is leaf, then goto step 2 otherwise repeat step 6.

The algorithm terminates as soon as a Z-region is found, such that $\alpha \geqslant Z_{\text{addr}}(\text{QB}_{\text{up}})$.

The *neighbour first point* and *neighbour region* optimizations additionally reduce the total I/O costs, since some of the "intersected" inner nodes need not to be retrieved. Moreover, the optimizations use only the information stored on the path stack, so they do not bring another kind of I/O overhead.

4.1. Time complexity

Complexity of the `TestZRegionIntersection` is linear according to Z-address bit-length (see Section 5.3), i.e. it is $O(n \log(|D|))$. Had we compared the `TestZRegionIntersection` and, as declared, the `GetNextZaddress` operations, it is important to realize that they are both of the same complexity.[6] In other words, they are very cheap.

The algorithm retrieves k query-intersected Z-regions by single LIFO traversal. In each visited node, the `TestZRegionIntersection` is called at most $\log_2(c)$-times (using halving the interval). The overall time complexity is the same as the one of Bayer–Markl's algorithm, i.e. it is $O(k \cdot h \cdot n \log(c) \log(|D|))$.

4.2. I/O costs

During the DRU algorithm running, each query-intersected node is retrieved only once, which is a consequence of single traversal through the UB-tree. Moreover, some inner nodes (which have to be accessed by simple UB-tree traversal) are even not retrieved due to the introduced optimizations performing "right-jumping" (steps 4 and 5 in the algorithm). The I/O costs are thus $I/Os \leqslant k \cdot h$.

[5] Conversions of data tuples (stored as Z-addresses) to the cartesian system are not necessary, see [11].

[6] Moreover, their complexity is the same as an equality test of two Z-addresses.

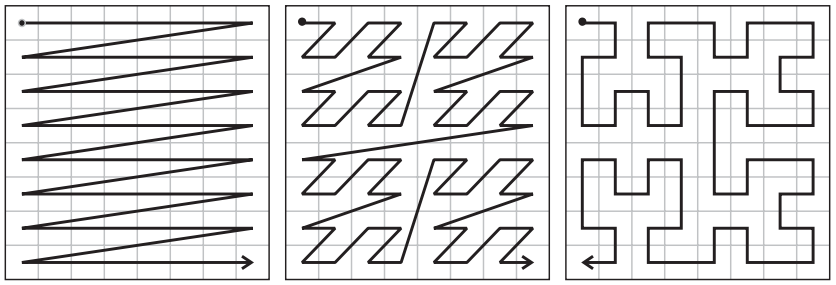


Fig. 7. Space filling curves.

5. Z-region intersection

In this section we discuss some properties of the Z-curve, which will help us in understanding the shape of Z-region. Using such information, we can design an algorithm for testing intersection between Z-region and query box.

5.1. Geometric properties of the Z-curve

In the vector space Ω a space filling curve (SFC) defines linear order for all the points in the space. In Fig. 7 three SFCs are depicted. Note that discrete SFC is just a geometric interpretation for linear ordering of points in the space and vice versa. For more details about SFCs we refer to Sagan [9].

For implementation of the basic UB-tree operations (i.e. insertion, deletion, point query) there is no qualitative reason why to choose just the Z-curve or another specific SFC. A particular reason for utilizing the Z-curve is the cheap algorithm of Z-address construction.

The important property of Z-curve is its high *locality*. The locality concept (for SFC) says that points that are "close" in the space (using some metric) are also "close" on the SFC (using the order). In other words, the Z-curve partially carries a topological information about the space; it locally preserves metric (we refer to Gotsman and Lindenbaum [15]). Alternatively, Markl [11] classifies the SFCs according to their *symmetry*—a self-similarity concept taken from fractal geometry.[7]

[7]For key retrieval using fractals see Faloutsos and Roseman [7].

5.1.1. 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

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

is called the Z-address of o.

The understanding value of the above formula seems to be quite low. Somewhat 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 o_i. In each step of the algorithm, the bits are "sliced" from the coordinates (one bit from each coordinate in each step) and "glued together" into a single bit-string. The algorithm of bit interleaving is depicted in Fig. 8.

However, we tried to investigate even more characteristics about the Z-curve's shape, especially those usable for the range query purposes. As a geometric framework, we took advantage of generalized quad-trees.

5.1.2. Hyper-quad trees

Generalized quad-trees and their modifications have been applied many times in the areas of CAD and GIS as well as in the area of SAM (see Samet [16]). However, in our approach we consider the generalized quad-tree a bit more abstractly, since

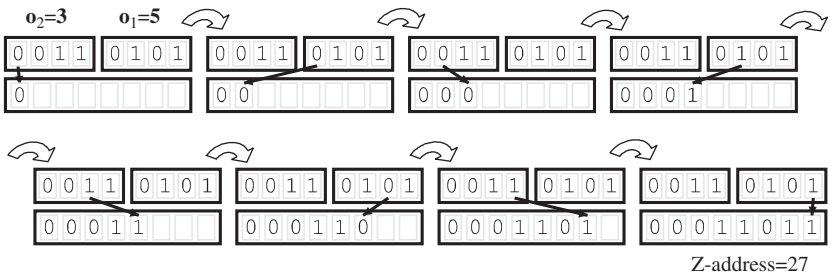


Fig. 8. Bit interleaving algorithm for two-dimensional point $o = [5, 3]$. $Z_{\text{addr}}(o) = 27$.

we use it just as a formal tool for studying the Z-curve.

In geometry, the term *quadrant* reflects an exactly defined quarter of *two*-dimensional space. Similarly, we can divide *single*-dimensional space into two halves, and in *three*-dimensional space we distinguish eight octants of space. Common to all these geometric constructs is a need to *partition* the space. Moreover, the partition is always performed using halving the space in all existing dimensions. We can generalize such information by definition of the hyper-quadrant of n-dimensional space.

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

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

The previous set-based definition does not cope with an identification of hquad according to 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 actually) that helps us to discover a relation between the hyper-quad trees and the Z-curve. Such hquad number—we call it *hquad code*—is constructed using successive halving of each dimension and testing whether the hquad is located either in the lower or in the upper half.

Definition 4 (*hquad code*). The *hquad code* is represented by a binary string, where each bit indicates the hquad's location according to one dimension. The ith bit is set to `0` if the hquad is located (according to the ith dimension) in the lower half of domain D, i.e. if $\text{HD}_i = \mathit{low}(D)$. The second case is dual, i.e. the ith bit is set to `1` if $\text{HD}_i = \mathit{up}(D)$.

The most significant bit (the left-most) indicates the location within the nth dimension, while the less significant bit (the right-most) indicates the location according to the first dimension. If we sort all the hquad codes lexicographically (i.e. left-to-right), we obtain an ordering of hquads. In Fig. 9 hquads and their codes are depicted. The bit-length of each hquad code is n, which is obvious from the code construction.

Definition 5 (*HQ-tree*). The *hyper-quad tree* (HQ-tree) is generalized quad-tree, representing a complete recursive partition of the n-dimensional vector space Ω. Each inner node of the HQ-tree contains a covering hquad and 2^n links to all its sub-hquads. The covering hquad of the root node is the whole space Ω, while the HQ-tree leaves represent all the points of Ω. The links to the sub-hquads are stored in ascending order, according to their hquad codes.

Corollary 2. *Since the domain D is a finite set, the height h of HQ-tree is $h = \log_2(|D|)$. Hquads in the leaves of HQ-tree are points, i.e. the domains of the deepest hquads contain single element. An arbitrary point of the space Ω is always hquad located in a leaf of the HQ-tree.*

Examples for 1D, 2D and 3D spaces are the binary tree, the quad-tree, and the octant-tree, see Fig. 10.

Finally, we need to establish unique identification of hquad within the HQ-tree hierarchy. This can be managed using the downward *navigation traversal* through the HQ-tree. The HQ-tree is traversed along a single path, while the *hquad navigation code* is constructed by concatenation of codes belonging to the hquads on the path. The navigation code (of variable length) uniquely determines the hquad position in HQ-tree.

Definition 6 (*hquad navigation code*). Let us have an hquad HQ. The binary string $navi(HQ) = c_0 \cdot c_1 \cdots c_{l-1}$ is called the *hquad navigation code* of HQ, if each substring c_i is code of such hquad at the ith HQ-tree level, that spatially contains HQ.

Corollary 3. *The larger hquads have shorter navigation codes and vice versa. The bit-length of navigation code for a point pt is* $|navi(pt)| = n\log_2(|D|)$.

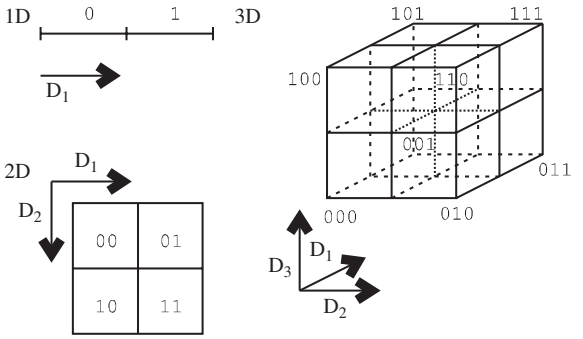


Fig. 9. Hyper-quadrants and their codes.

In Fig. 11 the navigation code construction in shown.

In the following we discuss the relation between the HQ-tree and the Z-curve.

Theorem 1. *The navigation code* $navi(o)$ *of* $o \in \Omega$ *is equivalent to the Z-address* $Z_{\text{addr}}(o)$ *of* o, *i.e.* $navi(o) = Z_{\text{addr}}(o)$.

Proof. If we realize, both of the Z-address construction algorithms, i.e. the navigation code construction and the bit interleaving, combine the bits of the source coordinates exactly the same way (see Figs. 8 and 11). □

Another relation between the Z-curve and the HQ-tree is hidden in the LIFO HQ-tree traversal. This traversal visits the HQ-tree leaves (points in space) in the same order as the Z-curve fills the space. Hence, we might sketch a simple recursive algorithm for drawing the Z-curve using HQ-tree traversal. As a side effect, such a drawing algorithm might be utilized as an efficient tool for Z-ordering of the whole universe Ω. This could be useful in various computer science disciplines, where every point of space carries some information (e.g. in image processing or pattern recognition).

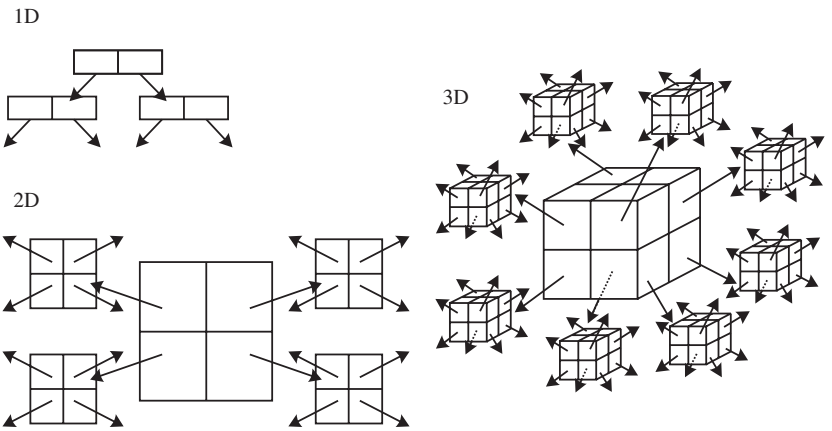


Fig. 10. HQ-trees—binary tree, quad-tree, and oct-tree.

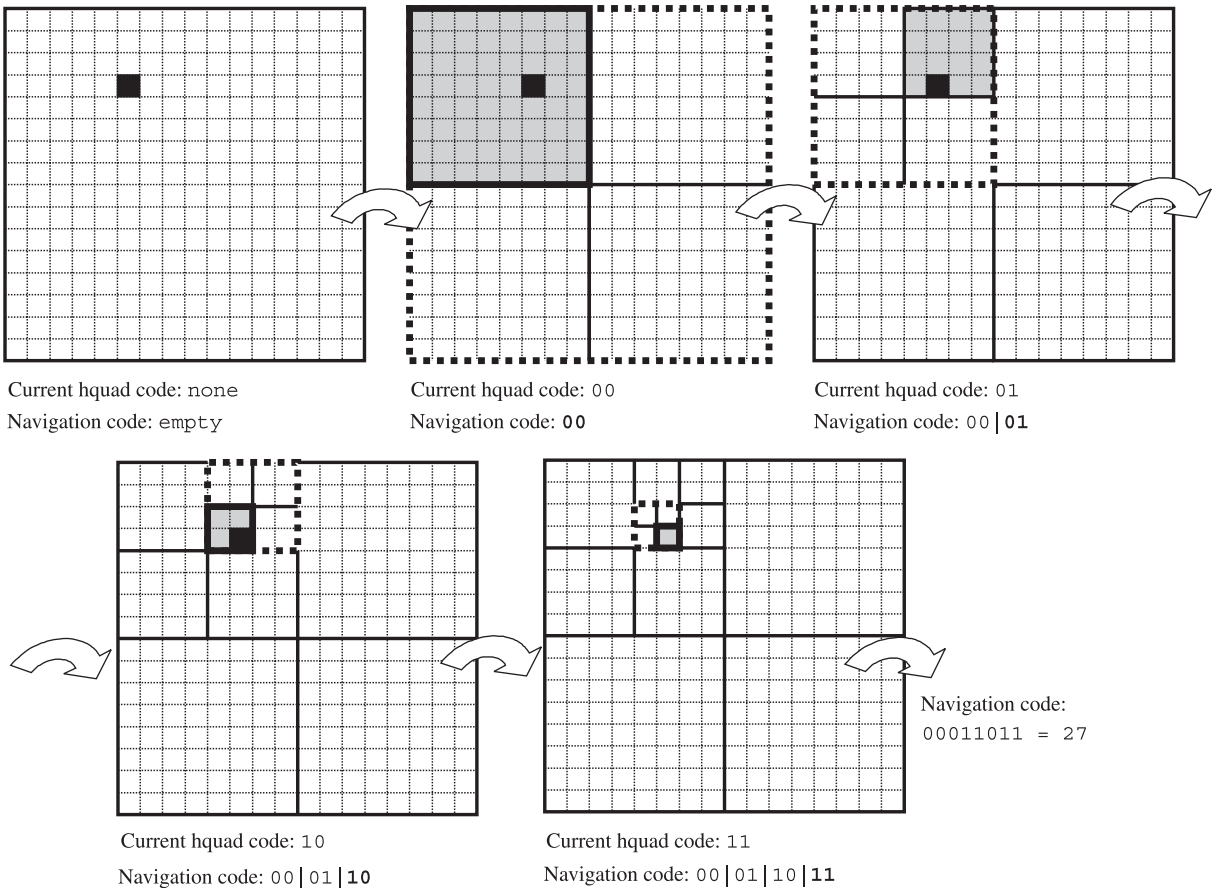


Fig. 11. Navigation code construction for point hquad located at coordinates [6,2].

5.1.3. Important consequences

Let us summarize several consequences that are important for further analysis. The consequences can be also observed in Fig. 12, where the Z-ordering of hquads at the first and the third HQ-tree level is depicted.

- The hquads define Z-regions which exactly fit the hquads.
- The hquads at a given HQ-tree level are Z-ordered according to their navigation codes.
- Consider a hquad at kth level of an HQ-tree. The Z-curve visits its sub-hquads at the $(k+1)$th level one-by-one, so that the hquad is entirely filled before the Z-curve enters the next hquad at the kth level.
- If the space Ω (a hquad generally) is halved into two half-spaces according to the last (nth) dimension, the Z-curve visits all points of the first half-space before it enters the second half-space. Thus, the half-spaces themselves are Z-ordered.

5.2. Minimal Z-region Hquad Envelope

Before we describe the linear algorithm for testing intersection between the query box and the Z-region, we have to introduce concept of so-called *minimal Z-region hquad envelope*.

Definition 7 (*minimal Z-region hquad envelope*). Every Z-region can be assembled by hquads

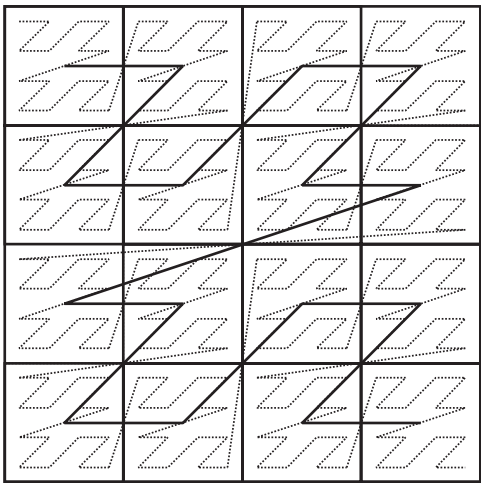


Fig. 12. Relationship between Z-curve and HQ-tree.

located at various levels of the HQ-tree. We call the set of hquads forming a given Z-region as *Z-region hquad envelope*. The envelope formed by the smallest number of hquads we call the *minimal Z-region hquad envelope*.

The minimal Z-region hquad envelope can be constructed by the following three-phase algorithm (see also an example in Fig. 13):

Algorithm 3 (*minimal Z-region hquad envelope*).

Input: Z-region $[\alpha : \beta]$
Output: set of hquads (minimal Z-region hquad envelope)

(1) Find the smallest hquad $hq_{\min}$ containing *both* Z-region bounds α and β.
(2) Send to the output all sub-hquads of $hq_{\min}$, which are located (according to hquad codes)

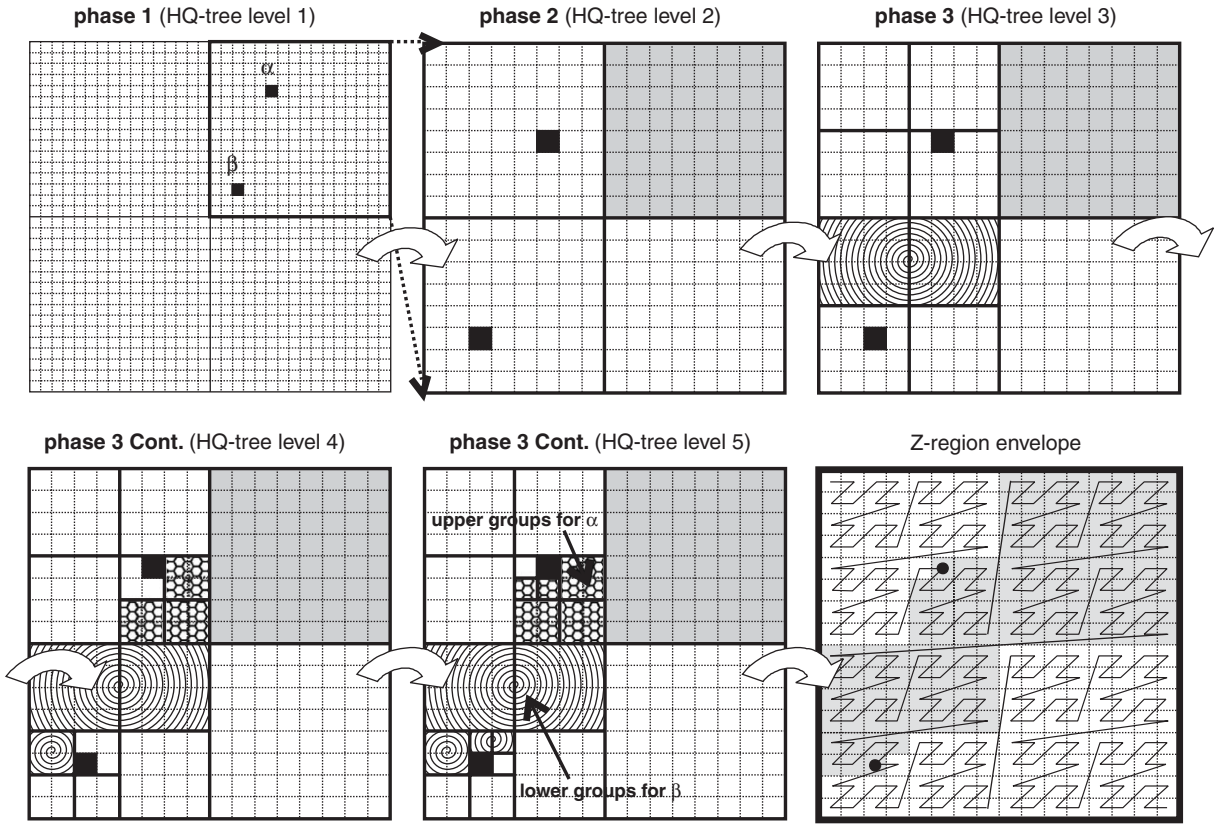


Fig. 13. Construction of the minimal Z-region hquad envelope. The Z-region $[\alpha : \beta]$ is formed by 13 hquads. For Z-region bounds α and β, the border hquads hq_α and hq_β are constructed in each step of the third phase.

entirely between the "border hquads" (which are two sub-hquads of $hq_{\min}$ containing α and β). Let us denote the border hquads as hq_α and hq_β.

(3) Process the following for both border hquads hq_α and hq_β. The hquad hq_α (hq_β) is decomposed into sub-hquads. The sub-hquad which contains the α (β) is set as the new border hquad hq_α (hq_β, respectively). Among the remaining sub-hquads, the *lower group* is formed by such sub-hquads, the codes of which are lower than code of hq_α (hq_β). Similarly, the *upper group* is formed by sub-hquads having their codes greater than code of hq_α (hq_β). In case of hq_α (hq_β) processing, the upper group (lower group respectively) is sent to the output. The steps of the third phase are repeated until hq_α and hq_β become points.

In the presented construction, we have assumed that every minimal Z-region envelope must contain at least two point hquads. However, there also exist Z-regions the minimal envelopes of which consist of hquads located at higher HQ-tree levels only. For example, envelope of the whole space Ω is single hquad. Such "rough" Z-regions can be handled in the second and the third phase of the algorithm but, for the sake of simplicity, we can omit such cases.

If we replace the "send hquad to the output" operation with the "test hquad and query box intersection" operation, we obtain the desired functionality of testing intersection between the Z-region and the query box.

From another point of view, we can imagine the envelope as a subtree in the HQ-tree. The Fig. 14 presents such a subtree for the envelope from Fig. 13. In the 3rd phase (levels 2–4), left branch (for the α bound) and right branch (for the β bound) are traversed down. In the left branch, the upper groups of sub-hquads are sent to the output (the greyed parts), while in the right branch, the lower groups of sub-hquads are greyed and thus sent to the output.

5.2.1. Intersection of Hyper-boxes

The "test hquad and query box intersection" operation is evaluated as an intersection of two hyper-boxes. As we can see in Fig. 15a, two hyper-boxes are intersected just in case that their ranges intersect in *all* dimensions. For ranges of a

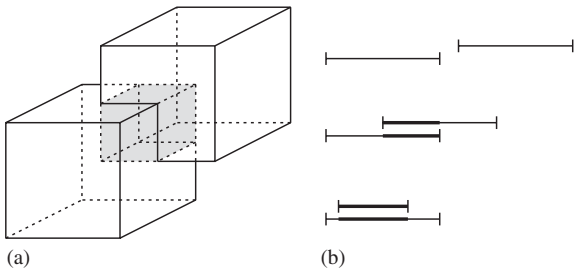


Fig. 15. Intersection of two hyper-boxes.

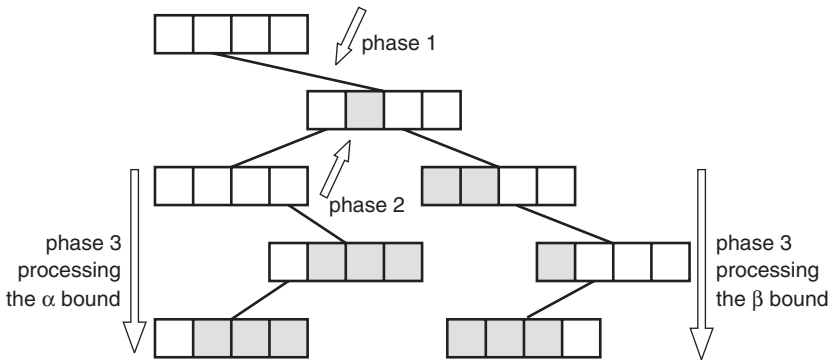


Fig. 14. The hquad envelope subtree.

particular dimension three cases may happen, see Fig. 15b.

If we denote the first hyper-box range of the ith 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 indicating whether two ranges are intersected:

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

Finally, two hyper-boxes are intersected if the statement holds for all i.

5.2.2. Time complexity

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

In the third phase, there are four nested loops. In the first loop, the HQ-tree is traversed down; the height of HQ-tree is $h = \log_2(|D|)$. In the second loop, there is up to $2^n - 1$ sub-hquads sent to the output (at each HQ-tree level). Finally, third loop and fourth loop are hidden in the intersection test of two hyper-boxes (testing n coordinates, each consisting of $\log_2(|D|)$ bits); i.e. it is $O(n \log(|D|))$. The overall time complexity is $O(n \cdot 2^n \log^2(|D|))$.

Had we used vector space of a small dimensionality, say $n < 10$, such complexity (exponential with n) would be acceptable. However, for high-dimensional spaces this algorithm is inefficient. Fortunately, there exists a solution reducing the complexity to linear according to Z-address bit-length, as we propose in the next section.

5.3. Linear intersection algorithm

The greatest component of the envelope construction/intersection 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, the lower group (upper group, respectively) can be spatially represented by at maximum n general hyper-boxes (not by hquads anymore). The idea is based on elimination of all dimensions during processing of the border hquad code. We present the idea for construction of the *reduced upper group*; the *reduced lower group* is constructed similarly.

Algorithm 4 (*reduced upper group construction*).

(1) The parent hquad is set as the actual hyper-box. Determine the border sub-hquad (with respect to the Z-region bound).
(2) The bits of the border sub-hquad code are iteratively read from the most significant to the less significant. We already know that each bit of the code determines location of the hquad according to the appropriate dimension.
    In each step, we process the next bit of the border hquad code. For each bit value, two cases may occur:
    (a) If the bit is set to `0`, it means that the border hquad is located in the lower half of the appropriate dimension. Since we construct the upper group, we send the upper half (according to the dimension) of the actual hyper-box to the output. The actual hyper-box is then set to its own lower half.
    (b) If the bit is set to `1`, it means that the border hquad is located in the upper half of the appropriate dimension. Since we construct the upper group, we ignore the lower half. The actual hyper-box is set to its own upper half.

    The steps are repeated until all bits, i.e. dimensions, are processed.

See examples in Fig. 16.

Corollary 4. *The algorithm constructs maximally n hyper-boxes for the reduced upper/lower group. In consequence, each hyper-box in the reduced group covers up to* 2^{n-1} *hquads* (*present in the appropriate unreduced group of hquads*). *Moreover, each hyper-box in the reduced upper/lower group is equal to union of* all *the remaining hquads located in upper/lower half of the divided dimension* (*remember the consequences summarized in Section* 5.1.3), *thus the hyper-box itself is a hyper-rectangular Z-region.*

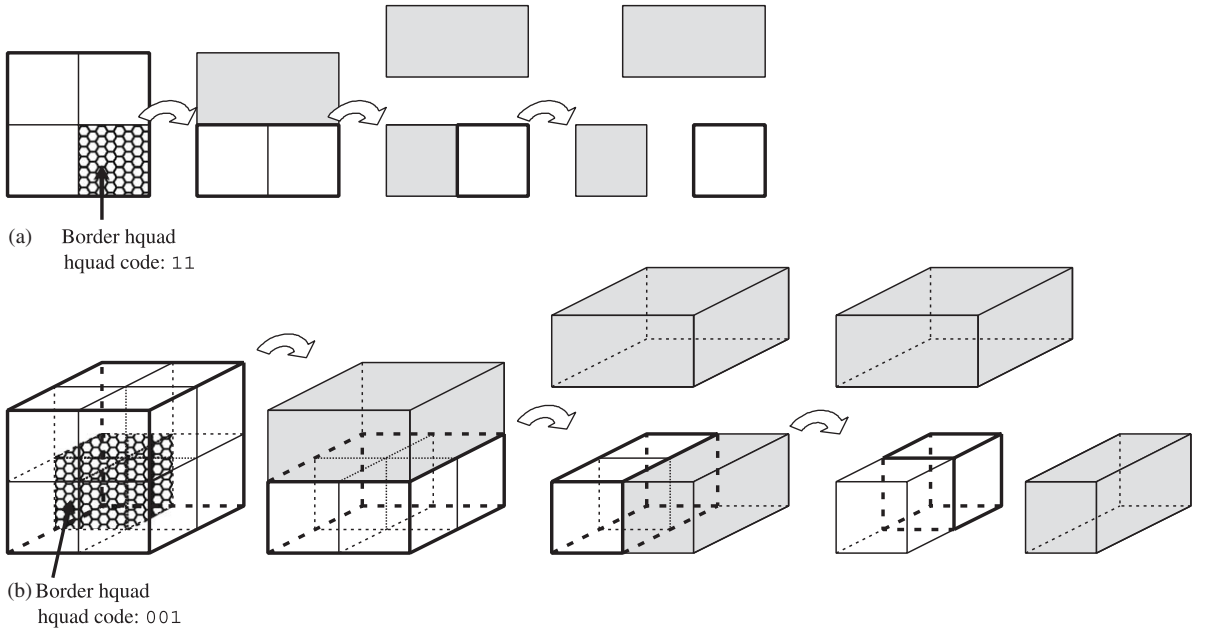


Fig. 16. (a) 2D example of reduced lower group construction. (b) 3D example of reduced upper group construction.

We use the particular reduced lower/upper group constructions for the linear, two-phase version of the intersection algorithm. As in the exponential case, we can interpret the intersection algorithm as a construction of Z-region envelope.

Algorithm 5 (*Z-region hyper-box envelope*).

Input: Z-region $[\alpha : \beta]$
Output: set of hyper-boxes (hyper-box envelope)

(1) The first phase is the same as by the exponential algorithm. The smallest hquad entirely containing the Z-region is determined. Further processing is restricted to this hquad.
(2) The second phase constructs the so-called *lower* and *upper* Z-region *half-envelopes*. The lower half-envelope is created for the upper bound β, while the upper half-envelope is created for the lower bound α. At each level of HQ-tree, the half-envelopes are consecutively constructed using the above described reduced upper/lower group construction. In order to keep the half-envelopes disjunct, only such hyper-boxes are added to the lower (upper) half-envelope, which do not contain β (α, respectively).

The geometric union of the two half-envelopes produces a "hyper-box envelope" spatially matching the "hquad envelope" (see proof at the end of this section). This fact also states that if the query box intersects a Z-region, then it must also intersect at least one of its half-envelopes, and vice versa.

As by the exponential algorithm, to obtain the intersection algorithm, we replace the "send hyper-box to the output" operation with the "test hyper-box and query box intersection" operation. Since the linear version of the intersection algorithm is a bit more complicated, we present it in a more explicit (C++-like) pseudo-code:

Algorithm 6 (*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
  i = 0
  While(alpha[i] = beta[i]) {
    spaceBox = spaceBox.GetSubHquad(alpha[i])
    i = i+1
  }
  // phase 2 - intersection of half-envelopes
  If (TestUpperHalfEnvelope(spaceBox, queryBox, alpha, beta) Or
      TestLowerHalfEnvelope(spaceBox, queryBox, alpha, beta)) Then
        return true
  Else
        return false
}

bool TestLowerHalfEnvelope(Hbox actualHBox, Hbox queryBox, Zaddress alpha, Zaddress beta)
{
  flag = false // keeps both half-envelopes disjunct
  // HQ-tree downward traversal
  For i = 0 To log2(|D|) − 1 {
    For j = n-1 To 0 { // process all dimensions
      If(alpha[i].GetBit(j)) Then {
        // set the actualHBox to its upper half
        actualHBox = actualHBox.GetUpperHalf(j)
      }
      Else {
        // test the intersection with the querybox
        If (flag And TestQueryBox(queryBox, actualHBox.GetUpperHalf(j))) Then return true
        If (TupleInsideBox(beta, actualHBox.GetUpperHalf(j)) Then flag = true
        actualHBox = actualHBox.GetLowerHalf(j)
      }
    }
  }
  // test the last point (alpha)
  return TestQueryBox(queryBox, actualHBox)
}
```

Notes to the pseudo-code:

- `TestUpperHalfEnvelope` is dual to `TestLowerHalfEnvelope`.
- `TupleInsideBox(Zaddress, qb)` examines whether the `Zaddress` lies inside the query box `qb`.
- `TestQueryBox(hbox, qb)` examines whether the `hbox` intersects `qb`.
- `alpha[i]`, `beta[i]` returns the *i*th hquad code of a Z-address.
- `topHquad.GetSubHquad(i)` returns the sub-hquad of `topHquad` identified by the hquad code *i*.

The linear algorithm (hyper-box envelope construction respectively) is depicted in Fig. 17. Note that hyper-boxes containing the opposite Z-region

bound cannot be sent to the output (see the light-grey boxes). This restriction (in the pseudo-code provided using the boolean `flag` variable) ensures that both constructed envelopes are disjunct.

Theorem 2. *For a given Z-region $Z = [\alpha : \beta]$, the two following statements hold*:

(a) *The minimal hquad envelope of Z exactly matches Z, i.e. the envelope determines the same set of points in vector space as the Z-region.*
(b) *The hyper-box envelope of Z exactly matches the minimal hquad envelope of Z (and thus matches Z).*

Proof. (a) The minimal hquad envelope is created by recursive decomposition of the space, such that hquads (their navigation codes) lie inside Z (within interval $\langle\alpha, \beta\rangle$, respectively). Due to recursion the decomposition of Z's space is complete.

(b) According to Corollary 4, the hyper-boxes in reduced upper/lower groups are hyper-rectangular Z-regions, thus the half-spaces decomposed into lower and upper half-envelopes are, mutually, Z-ordered. More specifically, there exists a ''border Z-address'' γ separating the two half-spaces. Simultaneously, the lower half-envelope is constructed for Z-region $[\alpha : \gamma]$, while the upper half-envelope is constructed for Z-region $[\gamma + 1 : \beta]$. In other words, the half-envelopes are disjunct and, simultaneously, there is no ''room'' between the upper and lower half-envelopes. Thus, union of the half-envelopes exactly matches the minimal hquad envelope and, in turn, the Z-region. □

5.3.1. Time complexity

The complexity of the linear intersection algorithm is $O(n \log(|D|))$. There are only two nested loops. The first one is the HQ-tree downward traversal, while in the second one all dimensions of a border hquad code are processed. Note that operations `TupleInsideBox` and `TestQueryBox` can be performed in $O(1)$ time, since the actual hyper-box is always halved in a single dimension in each step. Consequently, whether a tuple is inside the query box (whether a hyper-box is intersected by the query box, respectively) can be checked with respect to the respective dimension only. Hence, the overall time complexity of the algorithm is linear with the Z-address bit-length.

5.3.2. Relationship with the query box decomposition

The construction of the hyper-box envelope is an inverse technique to the query box decomposition approach, presented in Section 3.1. Instead of a set of Z-regions representing an envelope for the query box, we create sets of hyper-boxes representing envelopes for multiple Z-regions.

The query box decomposition is performed only once, while the set of interval queries can be processed by any indexing structure providing linear ordering of keys. Unfortunately, the size of the set of intervals is exponentially large (according to the dimensionality).

On the other side, the DRU algorithm computes the Z-region envelope (the Z-region intersection respectively) multiple times—for Z-regions ''potentially'' intersected by the query. However, the Z-region intersection is of linear complexity, so that the DRU algorithm is applicable, unlike the query box decomposition, also for high-dimensional spaces.

6. Experimental results

We made a set of experiments[8] with synthetic datasets of increasing dimensionality. The data tuples were generated into uniformly distributed clusters of a fixed radius (using the L_2 metric, see Fig. 18 for the 2D case) and indexed using the UB-tree. The number of tuples was increasing with the number of dimensions. We have examined the DRU algorithm and the Bayer–Markl's algorithm in the experiments.

In the table below see some statistics about the datasets as well as the created UB-tree indices.

[8] A comparison of UB-tree against other multi-dimensional structures was out of scope of this paper, we refer to [4,11] where the superiority of UB-tree (over R-tree, Grid-files, etc.) was already verified.

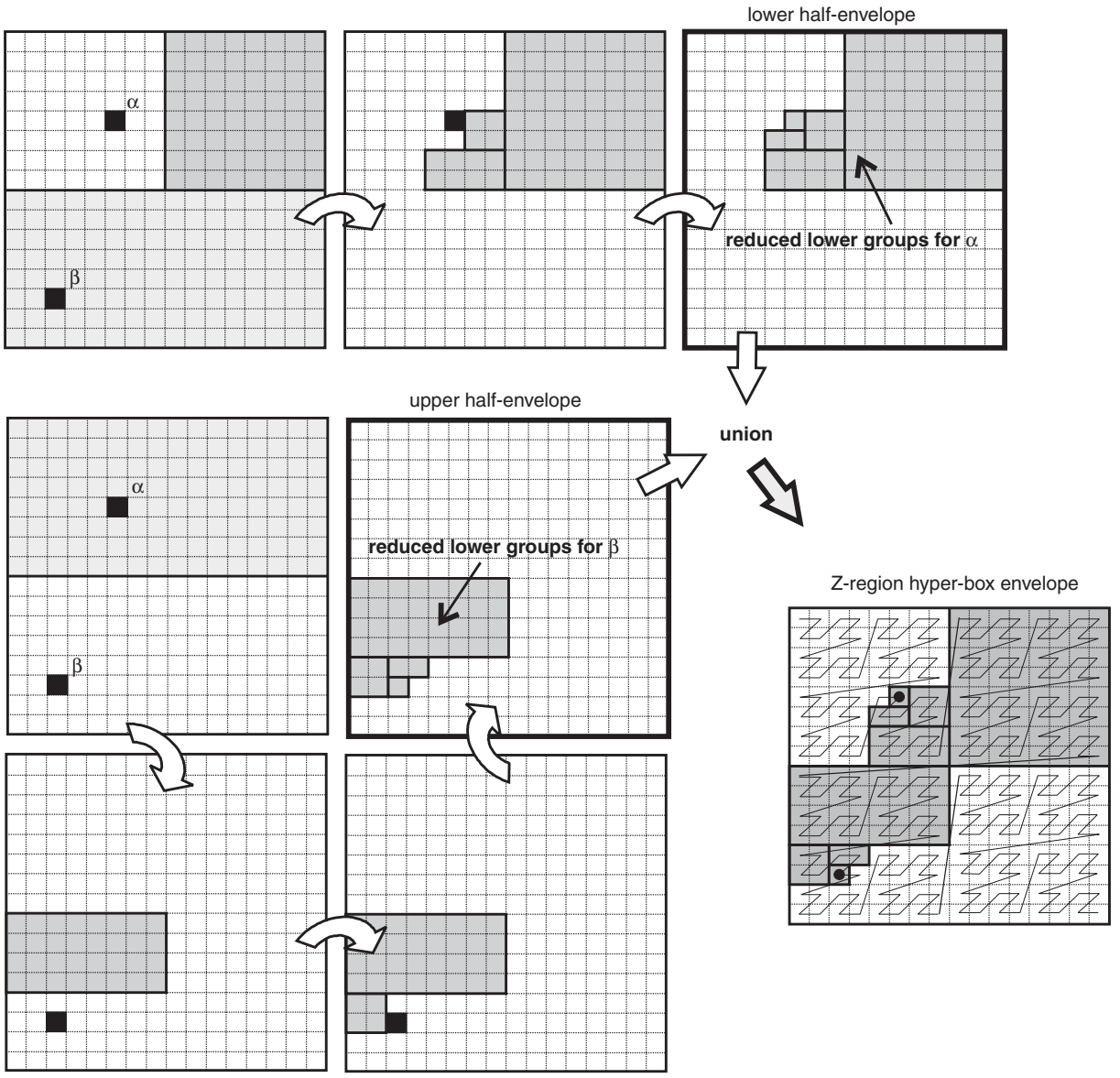


Fig. 17. The linear intersection algorithm (construction of Z-region hyper-box envelope). The envelope consists of 9 hyper-boxes, i.e. max. 9 hyper-boxes are tested for an intersection with the query box.

UB-tree statistics:

$\|D\|$	2^{32}	Dimensionality	2–30
Inserted tuples	524,288–7,864,320	Tree height (h)	4
Nodes	22,400–321,885	Z-regions (leaves)	21,475–321,885
Node capacity	35	Node utilization	69.7–69.8%
Node size	580–4612 bytes	Index file size	12.4 MB–1.44 GB

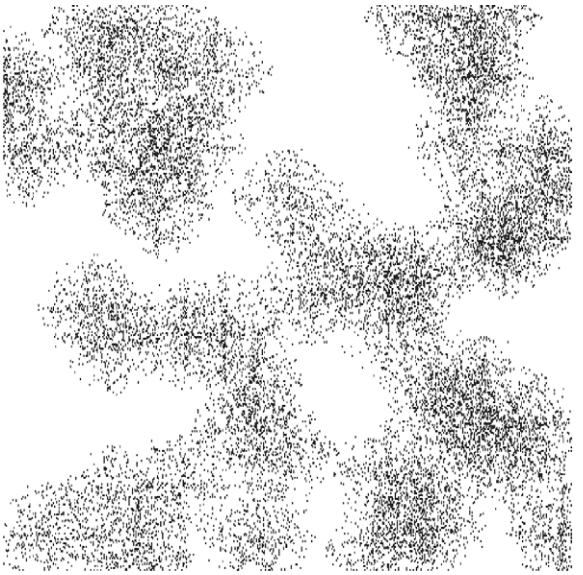


Fig. 18. Two-dimensional data set distribution.

We have generated from 24 to 120 query boxes for the experiments (the number of queries was increasing with the number of dimensions). The query boxes (of various shapes) were distributed randomly with respect to the distribution of data tuples. With the growing dimensionality the ranges of query boxes were fixed, thus the query box volumes were increasing, but the query box volume/space volume ratio was decreasing. Such a query box construction is typical for multi-dimensional applications. The results of range query experiments were averaged.

In Fig. 19a the range query selectivity is presented. The number of returned tuples is approximately constant, but the number of accessed Z-regions (retrieved leaves respectively) rapidly grows with the increasing dimensionality. The Fig. 19b shows real times of the DRU algorithm running.[9]

In Fig. 20 the original Bayer–Markl's algorithm (denoted as B–M algorithm) and the DRU algorithm are compared. Fig. 20a shows the number of disk page retrievals[10] (I/O costs), while Fig. 20b shows the number of computations spent by range query processing. As a single computation we generally consider each performed operation, which is of linear complexity according to the Z-address bit-length. Such operations are:

(1) comparison of two Z-addresses, testing Z-address inside Z-region, testing Z-address or tuple inside the query box,
   - utilized in **both** range query algorithms; for the *neighbour first point* optimization, for filtering tuples in leaves, for traversing the UB-tree, etc.

(2) computation of the `TestZRegionIntersection` operation,
   - utilized only in the **DRU** algorithm for the *neighbour region* optimization and for testing Z-region intersection while traversing the UB-tree,

(3) computation of the `GetNextZAddress` operation,
   - utilized only in the Bayer–Markl's algorithm for determination of the next intersecting Z-region.

Since all the operations are of the same complexity, we can use them as a computational unit, in order to compare both range query algorithms.

The results demonstrate that DRU algorithm works more efficiently, since the I/O costs and the computation costs are several times smaller than the costs spent by the Bayer–Markl's algorithm. The reason can be observed in Fig. 21, where the power of leaf optimizations is presented.

In Fig. 21a the successful optimization attempts are shown. A successful attempt of an optimization means that the respective Z-region was positively checked as query-intersected. The line labeled as "total attempts" denotes the total number of optimization attempts (successful as well as unsuccessful) for both leaf optimizations. The Fig. 21b shows the optimization effectiveness,

[9]Performed on an Intel Pentium ®4 2.4 GHz, 512 MB DDR333, Windows XP.

[10]In order to eliminate the influence of disk caching, we have considered only the logical disk accesses.

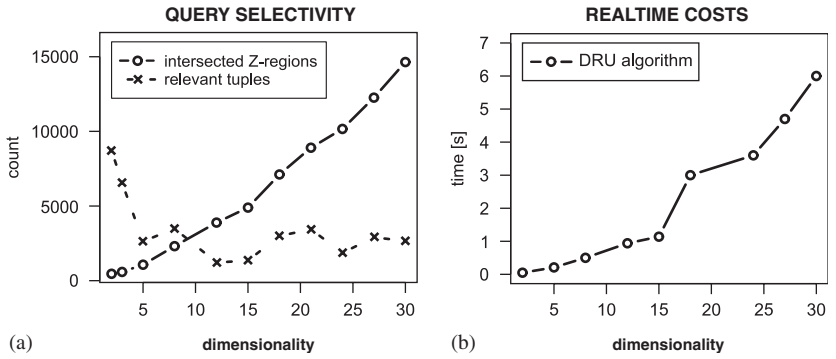


Fig. 19. (a) Query selectivity. (b) Realtime costs.

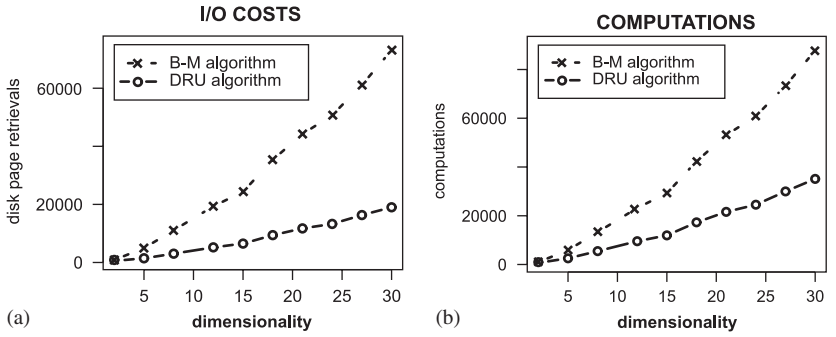


Fig. 20. (a) I/O costs. (b) Computations.

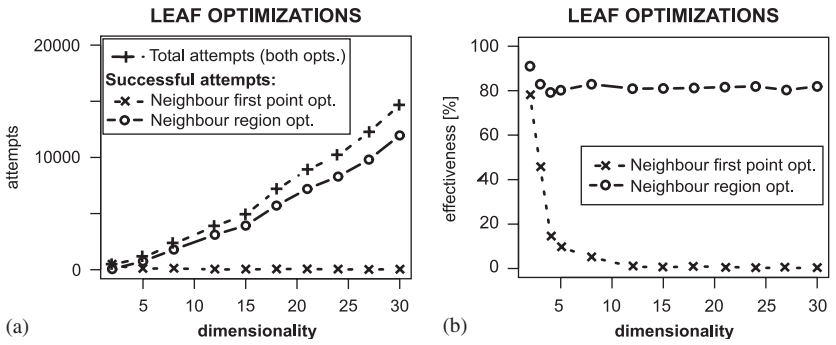


Fig. 21. (a) Leaf optimizations (successful attempts). (b) Leaf optimizations (effectiveness).

i.e. ratio of the number of successful attempts to the number of total attempts.

We can observe that for dimensionality $n = 2$ the *neighbour first point* optimization (used also in the Bayer–Markl's algorithm) has detected majority of the query-intersected neighbour regions. However, for $n > 4$ the *neighbour first point* optimization became almost useless. This was expectable, since the shape of Z-region is much more complicated for higher dimensionalities, so that the first point of a query-intersected Z-region is (almost) always outside the query box. However, the *neighbour region* optimization is more powerful, it works well even for high dimensionalities. In

fact, the effectiveness of the *neighbour region* optimization means that over 80% of right-neighbour-leaf's Z-regions were intersected by the query box.

As it has been mentioned in Section 4, a single successful optimization attempt saves up to h disk page retrievals needed for UB-tree traversal (a point query in case of Bayer–Markl's algorithm), it is sufficient to "jump" to the right neighbouring leaf. Consequently, from Fig. 21 we can reason out, that majority of the UB-tree inner nodes was not accessed due to the *neighbour region* optimization.

The above presented results bring us to the discussion about the influence of the *curse of dimensionality* [2,17] on range query processing in the UB-tree. With the growing dimensionality the costs grow as well, however, less than exponentially. In Fig. 22a we see a ratio of tuples inside the query box to the number of intersected Z-regions. The Fig. 22b shows a ratio of the *relevant Z-regions* to all of the query-intersected Z-regions accessed by the DRU algorithm. As a relevant Z-region we consider such Z-region, that contains at least one data tuple contained also by the query box. The ratio says that for higher dimensionalities, more than 95% of the query-intersected Z-regions "give" no tuples to the result. The reason is obvious—topological properties of the Z-curve become worse for higher dimensionalities, and the Z-region volumes become larger.

On the other side, the Fig. 22b shows also a ratio of query-intersected Z-regions (accessed by the DRU algorithm) to the Z-regions contained in the interval $[Z_{\text{addr}}(\text{QB}_{\text{low}}) : Z_{\text{addr}}(\text{QB}_{\text{up}})]$ (i.e. in interval of the minimal Z-region bounding the entire query box). One could expect that negative effects of the curse of dimensionality will raise this ratio up to 100%, which would be the same as a traversal over majority of the UB-tree nodes. However, this experiment shows that (even for high dimensionalities) the number of accessed Z-regions intersected by the query box is much smaller than the number of Z-regions within the above mentioned interval. This particular result indicates that UB-tree together with the DRU algorithm is quite resistant to the curse of dimensionality. For a comparison, the well-known *R-tree* [19](but also the *R*-tree* [18]) used in many applications is highly affected by the curse of dimensionality and its usage for high-dimensional indexing is nearly impossible.

7. Conclusions and outlook

In this paper we have proposed a new algorithm (called DRU algorithm) for range query processing in the Universal B-tree (UB-tree). The DRU algorithm utilizes an operation detecting intersection between Z-region and query box, which is used for a more efficient query processing. The Z-region intersection operation is of linear time complexity according to the Z-address bit-length.

The experimental results have shown that DRU algorithm makes the UB-tree suitable for efficient search in high-dimensional vector spaces. In particular, using DRU algorithm almost all of

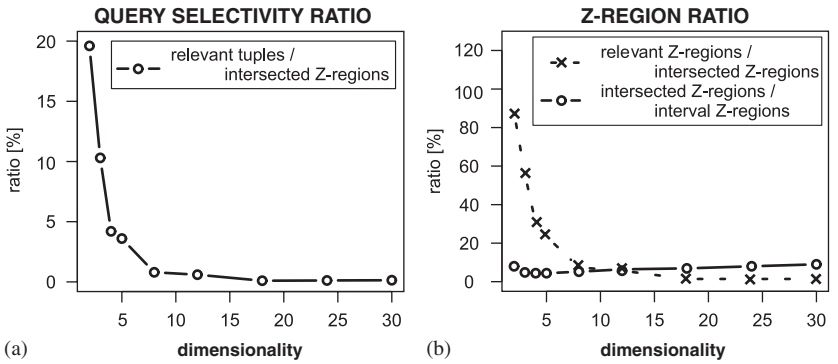


Fig. 22. (a) Query selectivity ratio. (b) Z-region ratio.

the UB-tree inner nodes need not to be accessed, which is a significant improvement when compared to the original UB-tree range query algorithm. Moreover, the DRU algorithm is fully applicable to the BUB-tree [3], a recent modification of the UB-tree.

In the future we would like to develop algorithms for (B)UB-trees, based on the Z-region intersection algorithm, e.g. algorithms for distance-based queries (spherical range queries and k-NN queries, in particular).

Acknowledgements

This research has been partially supported by grant No. GAČR 201/03/0912 of the Grant Agency of the Czech Republic.

References

[1] V. Gaede, O. Günther, Multidimensional access methods, ACM Comput. Surv. 30 (2) (1998) 170–231.
[2] C. Böhm, S. Berchtold, D.A. Keim, Searching in high-dimensional spaces: index structures for improving the performance of multimedia databases, ACM Comput. Surv. 33 (3) (2001) 322–373.
[3] R. Fenk, The BUB-tree, in: Proceedings of the 28th Conference VLDB, Morgan Kaufmann Publishers Inc., Los Altos CA, 2002.
[4] R. Bayer, The universal B-tree for multidimensional indexing: general concepts, in: Proceedings of the International Conference on Worldwide Computing and Its Applications, Springer, Berlin, 1997, pp. 198–209.
[5] J.A. Orenstein, T.H. Merrett, A class of data structures for associative searching, in: Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, ACM Press, New York, 1984, pp. 181–190.
[6] C. Faloutsos, Gray codes for partial match and range queries, IEEE Trans. Softw. Eng. 14 (10) (1988) 1381–1393.
[7] C. Faloutsos, S. Roseman, Fractals for secondary key retrieval, in: Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, ACM Press, New York, 1989, pp. 247–252.
[8] J. Orenstein, A comparison of spatial query processing techniques for native and parameter spaces, in: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, ACM Press, New York, 1990, pp. 343–352.
[9] H. Sagan, Space-Filling Curves, Springer, New York, 1994.
[10] C. Böhm, G. Klump, H.-P. Kriegel, XZ-Ordering: a space-filling curve for objects with spatial extension, in: Proceedings of the 6th International Symposium on Advances in Spatial Databases, Springer, Berlin, 1999, pp. 75–90.
[11] V. Markl, Processing relational queries using a multi-dimensional acces technique, Dissertations in Database and Information Systems-Infix, vol. 59, 1999, 217pp.
[12] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, R. Bayer, Integrating the UB-tree into a database system kernel, in: Proceedings of the 26th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers Inc., Los Altos, CA, 2000, pp. 263–272.
[13] J.K. Lawder, P.J.H. King, Using space-filling curves for multi-dimensional indexing, in: Proceedings of the 17th British National Conference on Databases, Springer, Berlin, 2000, pp. 20–35.
[14] J.K. Lawder, P.J.H. King, Querying multi-dimensional data indexed using the Hilbert space-filling curve, SIGMOD Rec. 30 (1) (2001) 19–24.
[15] C. Gotsman, M. Lindenbaum, On the metric properties of discrete space-filling curves, IEEE Trans. Image Process. 5 (5) (1996) 794–797.
[16] H. Samet, The Design and Analysis of Spatial Data structures, Addison-Wesley, Reading, MA, Longman, New York, 1990.
[17] C. Yu, High-Dimensional Indexing: Transformational Approaches to High-dimensional Range and Similarity Searches, Springer, New York, 2002.
[18] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The R^*-tree: an efficient and robust access method for points and rectangles, in: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, ACM Press, New York, 1990, pp. 322–331.
[19] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, ACM Press, New York, 1984, pp. 47–57.