The Buddy-Tree: An Efficient and Robust Access Method for Spatial Data Base Systems [*]

BERNHARD SEEGER[+] and HANS-PETER KRIEGEL
PRAKTISCHE INFORMATIK, UNIVERSITY OF BREMEN, D-2800 BREMEN 33, WEST GERMANY

Abstract
In this paper, we propose a new multidimensional access method, called the buddy-tree, to support point as well as spatial data in a dynamic environment. The buddy-tree can be seen as a compromise of the R-tree and the grid file, but it is fundamentally different from each of them. Because grid files loose performance for highly correlated data, the buddy-tree is designed to organize such data very efficiently, partitioning only such parts of the data space which contain data and not partitioning empty data space. The directory consists of a very flexible partitioning and reorganization scheme based on a generalization of the buddy-system. As for B-trees, the buddy-tree fulfills the property that insertions and deletions are restricted to exactly one path of the directory. Additional important properties which are not fulfilled in this combination by any other multidimensional tree-based access method are: (i) the directory grows linear in the number of records, (ii) no overflow pages are allowed, (iii) the data space is partitioned into minimum bounding rectangles of the actual data and (iv) the performance is basicly independent of the sequence of insertions. In this paper, we introduce the principles of the buddy-tree, the organization of its directory and the most important algorithms. Using our standardized testbed, we present a performance comparison of the buddy-tree with other access methods demonstrating the superiority and robustness of the buddy-tree.

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

1. Introduction

In non-standard database applications, such as geographic information processing or CAD/CAM, access methods are required that support efficient manipulation of multidimensional geometric objects on secondary storage. Moreover, efficient access methods are an essential part in knowledge-based systems [HCKW 90]. We can basicly distinguish between point access methods (PAMs) and spatial access methods (SAMs) which are designed to handle multidimensional point data, e.g. records ordered by a multidimensional key, and spatial data, e.g. polygons or rectangles, respectively.

First of all, these access methods must be dynamic, i.e. they should support arbitrary insertions and deletions of objects without any global reorganizations and without any loss of performance. Moreover they should efficiently support a large set of queries, such as range, partial match, join and nearest neighbor queries.

The basic principle of all multidimensional PAMs is to partition the data space into page regions, shortly regions, such that all records of a data page are taken from one region. We classify according to the following three properties of regions: the regions are pairwise disjoint or not, the regions are rectangular or not and the partition into regions is complete or not, i.e. the union of all regions spans the complete data space or not. Obviously, this classification yields six classes, four of which are filled with known PAMs. Without going into detail, in table 1 we present well known PAMs according to these three criteria.

All of the PAMs in class (C 1) perform rather efficient for uniform and uncorrelated data. However, for highly correlated data their performance degenerates.

(*) This work was supported by grant no. Kr 670/4-3 from the Deutsche Forschungsgemeinschaft DFG (German Research Society) and by the Ministry of Enviromental and Urban Planning of Bremen

(+) B.Seeger has a one year leave of absence from the University of Bremen which he presently spending at the University of Waterloo, Canada, financially supported by a Post Doctoral Fellowship from the DFG

class	property			PAM
	rectangular	complete	disjoint	
(C1)	X	X	X	interpolation hashing [Bur 83], MOLHPE [KS 86], quantile hashing [KS 89], PLOP-hashing [KS 88], k-d-B tree [Rob 81], multidimensional extendible hashing [Tam 82,Oto 84], balanced multidimensional extendible hash tree [Oto 86], grid file [NHS 84], 2-level grid file [Hin 85], interpolation-based grid file [Ouk 85]
(C2)	X	X		twin grid file [HSW 88]
(C3)	X		X	buddy tree, multilevel grid file [WK 85]
(C4)		X	X	B^+-tree with z-order [OM 84], BANG file [Fre 87], hB-tree [LS 89]

Table 1 : Classification of multidimensional PAMs.

Therefore other PAMs like the BANG-file or hB-tree have been proposed allowing more general shapes of regions which are constructed by difference and union of rectangles.

Quite a different approach for the efficient organization of highly correlated data is the buddy-tree. The most important characteristic is that the union of all regions does not span the complete data space. Thus the buddy-tree avoids partitioning empty data space. Instead the buddy-tree uses a similar concept as the R-tree [Gut84] and the R*-tree [BKSS 90] for spatial data, but differs from the R-tree variants by avoiding overlap in the tree directory. In comparison to previously proposed tree structures such as the K-D-B-tree, the buddy-tree guarantees a more efficient dynamic behavior.Moreover, indirect splits which cause low storage utilization and high insertion costs in the K-D-B-tree, are completely avoided. Therefore, the same properties are fulfilled as for B-trees [BM 72]:deletions, insertions and exact match queries are restricted to one path of the directory. This behavior is guaranteed by using a generalization of the buddy system which was originally proposed for the grid file. Due to this concept, the performance of the buddy-tree is almost independent of the sequence in which data is inserted.

Furthermore, we propose a special implementation technique for the buddy-tree which can be generalized to other access methods, such as the R-tree variants. From this the buddy-tree gains a high fan out of the directory nodes. Thus the height of the tree and the retrieval cost are reduced. Most SAMs assume that geometric objects are approximated by a minimal bounding rectangle whose sides are parallel to the axes of the data space. One technique to generate such a SAM from a PAM is the transformation of d-dimensional rectangles into 2d-dimensional points where for example the first d components represent the center, the remaining d components represent the extension of the rectangle ([Hin85], [SK88]). These 2d-dimensional points are highly correlated and occupy only a small part of the data space. In particular for such distributions the buddy-tree performs very efficiently.

In the paper we will use the following notations: The parameter d, $d \geq 1$, specifies the dimension of the data space D. The data space D is composed of the domains D_i, $1 \leq i \leq d$, of the i-th axis. On these domains an order relation should be well defined. Without loss of generality we assume that D is given by the d-dimensional unit square $[0,1)^d$. The parameters b, $b > 1$, and c, $c > 1$, denote the capacity of a data page and directory page, respectively.

The paper is organized as follows. In section 2 we introduce the principles and the properties of the buddy-tree on a more informal level. In section 3 we present a formal description of the structure of the buddy-tree and in section 4 we propose a generally applicable implementation technique for increasing the fan out of directory nodes. Section 5 contains a description of the essential algorithms of the buddy-tree. Finally, in section 6 we present an experimental performance comparison which demonstrates the superiority of the buddy-tree to other PAMs, such as the hB-tree, the BANG-file and the grid file.

2. The Principles of the Buddy-Tree

The buddy-tree organizes data using a tree-based directory where each axis is treated equally. In contrast to the K-D-B-tree [Rob81] (one of the first multidimensional trees), the buddy-tree performs well in a highly dynamic environment, i. e. insertions, deletions and a change of the data distribution do not affect performance. This property is achieved by applying a modified version of the so-called buddy-system which is well-known from the grid file [NHS84] to the buddy-tree. Additionally, the performance of the buddy-tree is almost independent of the sequence of insertions which is an essential drawback of previous tree-structures, like the K-D-B-tree or hB-tree [LS89].

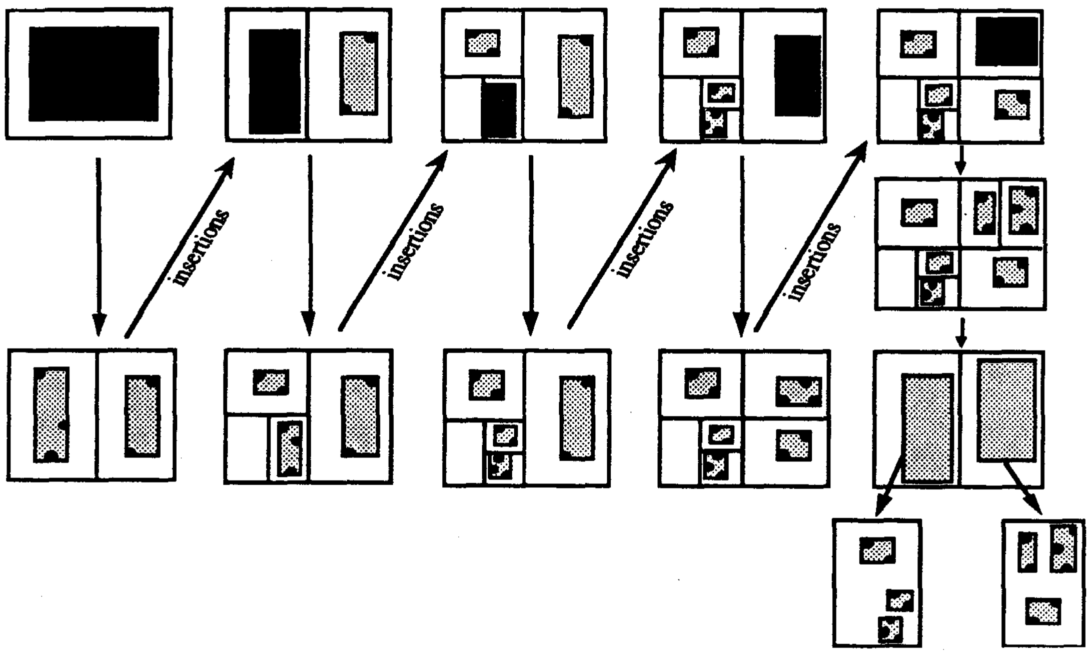
Another important feature of the buddy-tree is that it does not partition empty data space. Therefore queries, such as partial match queries, where the query region intersects with empty data space, can be performed much faster than by conventional structures partitioning the complete data space. This property is very similar to the variants of the R-tree, originally designed for spatial data. Con-trary to the R-tree, the buddy-tree does not allow overlap in the directory nodes and can therefore guarantee that insertions, deletions and exact match queries are restricted to one path of the directory. Additionally, we incorporate an implementation technique in the buddy-tree which in-creases the fan out of the directory nodes (see section 4).

The following catalogue summarizes the design properties of the buddy-tree:

- empty data space is not partitioned
- insertion and deletion of a record is restricted to exactly one path
- no overflow pages
- directory grows linear in the number of records
- performance is basicly independent of the sequence of insertions
- efficient behavior for insertions and deletions
- very high fan out of the directory nodes

With the following example we intend to visualize the basic properties of the buddy-tree:

Example 2.1:
Let the dimension be d = 2, the capacity of a directory page be c = 5 and the capacity of a data page be b = 4. Then the following snapshots depict the growth of the buddy-tree starting with the empty file. In the data pages the actual points are stored. Minimum bounding rectangles of at most 4 points are represented in the directory pages indicated by a light fill pattern. The white area corresponds to empty data space which is not managed by the buddy-tree (important design property). The first line in our example shows states of the buddy-tree with an overflowing data page depicted by a dark fill pattern. In the second line the corresponding subsequent state after the page split is depicted. The rightmost overflow of a data page implies an overflow of the one and only directory page resulting in a buddy-tree of height two.



Additionally to the above design properties the following technical properties can be seen from the above snapshots:

- partitions into minimum bounding rectangles of points and subrectangles in directory pages
- rectangles in directory pages are disjoint
- pointers are disjoint

Following these basic ideas the formal description of the structure of the buddy-tree and of its algorithms is presented in the next three sections.

3. Formal Description of the Buddy-Tree

The nodes of the tree-directory consist of a collection of entries $\{E_1, \ldots, E_k\}$, $k \geq 2$. Each entry E_i, $1 \leq i \leq k$, is given by a tuple $E_i = (R_i, p_i)$ where R_i is a d-dimensional rectangle and p_i is a pointer referring to a subtree or to a data page containing all the records of the file which are in the rectangle R_i. In this paper, a rectangle is always assumed to be parallel to the axis of the d-dimensional data space. In particular to support the dynamic behavior, the set of rectangles in a directory node must be a regular B-partition of the data space. An exact description of that condition is given by the following definitions.

Definition 1:
Given two d-dimensional rectangles R, S with $R \subseteq S$, R is called a B-rectangle of S, iff it can be generated by successive halfing of S.

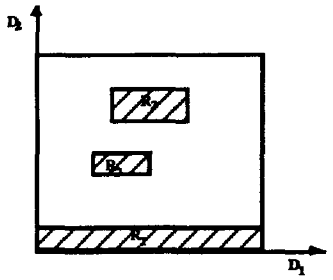


Figure 3.1: The rectangles R_1 and R_2 are B-rectangles of D and R_3 is not a B-rectangle of D.

In this definition the sequence of axes where halfing is performed is irrelevant. Notice that S may be the given data space D. In figure 3.1 we have depicted three rectangles R_1, R_2, R_3, where R_1 and R_2 are B-rectangles of the data space D and R_3 does not fulfil the property of a B-rectangle.

Properties of B-rectangles

1. If R, S are B-rectangles of the data space D and $R \subseteq S$, then R is a B-rectangle of S.
2. If A is a B-rectangle of D and we double D in direction of an arbitrary axis (naming the new data space D^2) then A is also a B-rectangle of D^2.
3. For an arbitrary rectangle $R \subseteq D$, there exists a smallest B-rectangle of D such that $R \subseteq B$. We call such a B-rectangle the B-region of R, short B(R). Such a B-region also exists for a union of rectangles $R_1 \cup R_2 \cup \ldots \cup R_k$, $k \geq 1$.

Definition 2:
A set of d-dimensional rectangles $\{R_1, \ldots, R_k\}$, $k \geq 1$, is called a B-partition of the data space D,
iff $B(R_i) \cap B(R_j) = \emptyset \quad \forall\, i, j \in \{1, \ldots, k\}, i \neq j$

Let (S, q) be an entry of an inner node of the buddy-tree where q refers to the node $\{(R_1, p_1), \ldots, (R_k, p_k)\}$, $k \geq 2$. Then we require that the set of rectangles $\{R_1, \ldots, R_k\}$ is a B-partition of B(S). From definition 2 it follows that if $\{R_1, \ldots, R_k\}$ is a B-partition, all sets $\{T_1, \ldots, T_k\}$ are B-partitions where $R_i \subseteq T_i \subseteq B(R_i)$, $1 \leq i \leq k$. In particular $\{B(R_1), \ldots, B(R_k)\}$ is a B-partition, also called the maximum B-partition. Which B-partition should be used in an implementation is discussed in a later section.

An important feature of a multidimensional access method is its efficient dynamic behavior. To obtain that, it must be possible to merge without destroying the order preservation. The buddy-tree merges two pages, if the resulting partition in the father node is again a B-partition. For this, the regions of the pages must be buddies, which is formalized in definition 3.

Definition 3:
Let $V = \{R_1, \ldots, R_k\}$ a B-partition, $k > 1$, and let $S, T \in V$, $S \neq T$. The rectangles S, T are called buddies,
iff $B(S \cup T) \cap B(R) = \emptyset \quad \forall\, R \in V \setminus \{S, T\}$

An important criterion for the efficiency of the dynamic behavior is the number of possibilities for a merge. In case of the buddy-tree this results in the question how many buddies exist. Let us first mention that in case of the grid file the maximum number of merge candidates is d, whereas for the K-D-B-tree only regions which result from a split are allowed to be merged, i. e. there is only one candidate for a merge. Before we present a bound for the buddy-tree, we need the definition of the level of a B-partition.

Definition 4:
Let V be a B-partition and $length_i$ (R) the length of the segment Π_i (B(R)), R ∈ V, $1 \le i \le d$, where Π_i (S) is the projection of the rectangle S ∈ D onto the i-th axis.
The local level of the i-th axis is given by

$$lev_i := \max_{R \in V} \log_2 (length_i(D) / length_i(R))$$

Let z denote the axis with the highest local level.
Then the level L of the B-partition is given by

$$L := d (lev_z - 1) + z.$$

Theorem 1:
Let lev > 0, lev_i = lev for $1 \le i \le d$, and V be a B-partition of level L = lev * d. Then the maximum number M of buddies for an arbitrary region R ∈ V is $\mathbf{M \ge \min \{|V| - 1, d + (lev - 1) d (d - 1)/2\}}$

Proof.: see [See 89]

The efficient dynamic behavior of the buddy tree results from a considerably higher number of candidates for performing a merge operation, as it can be seen from the following table 3.2:

PAM	number of candidates for a merge
hB-tree	0
K-D-B-tree	1
grid file	d
buddy-tree	> d

Table 3.2

Another serious problem of the original K-D-B-trees, also occurring in R^+-trees [FSR 87], is that a split of a directory page may produce an indirect split of subtrees. In figure 3.3 we have depicted a B-partition of a directory node. If we assume a directory capacity of 4, this node must be split into two which is performed by finding an axis and a hyperplane perpendicular to this axis dividing the directory entries into two disjoint sets of directory entries. This could not be achieved without cutting a rectangle in the directory page. A first approach suggested for the K-D-B-tree is to split all nodes belonging to such an intersected region into two, resulting in a possibly low storage utilization. Obviously, storage utilization will get out of control. Another drawback is that a split and therefore an insertion is not anymore restricted to one path of the tree. For the buddy-tree we avoided these drawbacks by allowing only a special class of B-partitions, called regular B-partitions.

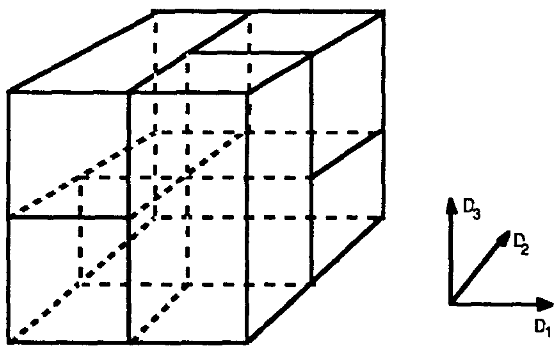


Fig. 3.3: A B-partition is given for which each of the hyperplanes cuts a B-region

Definition 5:
Let V = $\{R_1, \dots, R_k\}$, $k \ge 2$, be a B-partition. V is called regular, iff all B-rectangles B (R_i), $1 \le i \le k$, can be represented in a kd-trie.

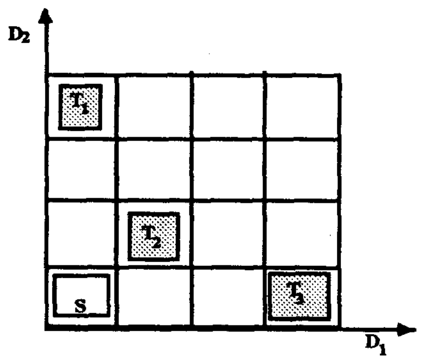


Fig. 3.4: For d=2 and L=5, we have illustrated the buddies T_1,T_2 and T_3 of S

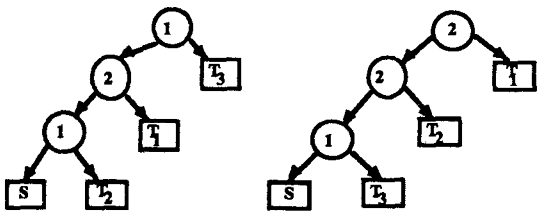


Fig. 3.5: Two kd-trie representations of the B-partition of fig. 3.4

A kd-trie [Ore82] is a binary digital tree where the internal nodes consist of an axis and two pointers referring to subtrees. In the leaves of the tree the rectangles of a B-partition are represented. Each internal node represents a B-rectangle and the root represents the complete region. In the left or right subtree of such a node, all rectangles are represented which are in the left or right half of the B-rectangle with respect to the corresponding axis, respectively. On the left hand side of figure 3.5 we have depicted a kd-trie corresponding to the B-partition of figure 3.4. Let us mention that there is no unique kd-trie representation of a B-partition. For example, the kd-trie on the right hand side of figure 3.5 represents also the B-partition of figure 3.4.

Considering regular B-partitions, we can also find a split axis which does not intersect with any rectangle of the B-partition. This can be done by using one of the axes denoted in the root of the kd-tries. The test, whether a B-partition is regular, costs quite a bit CPU-time and should not be performed often. The buddy-tree uses the test,if and only if two pages should be merged. After a split this test does not need to be executed, because in such a situation a leaf of the corresponding kd-tries is split into two and an internal node is added to the tree structure referring to these leaves. Let us mention that the grid file uses a very similar concept for detecting deadlocks. However, in case of the buddy-tree deadlocks cannot occur, because empty data space is not represented, i. e. rectangles without containing a record are not represented in the directory.

4.Increasing the Fan Out of the Directory Nodes

One shortcoming of the buddy-tree as well as of the R-tree is the relatively low fan out of the directory nodes, because both structures store sets of d-dimensional rectangles in their directory nodes. In this section, we suggest a representation of the rectangles which is similar to that of the so-called hash-trees ([Oto86], [Ouk85]). The basic idea is to use a d-dimensional orthogonal grid with a dynamicly varying resolution for each node. Only those rectangles are accepted which can be exactly mapped onto such a grid. These rectangles are represented by two cells matching the lower left and upper right corner of the rectangles. The cells are addressed using a hashing function. Therefore, instead of two d-dimensional points, only two hash values are necessary for the representation of the rectangles. Thus the fan out will increase. For the representation of rectangles by hash-values we decided to use z-values [OM83].

Another important characteristic of the buddy-tree is that the grid belonging to a directory node does not partition the minimal bounding rectangle M into equal sized cells, but partitions the B-region of M. The partition of the minimum bounding rectangle by a grid would lead to severe problems. If we merge two nodes, a completely new computation of the new grid must be performed. More seriously, we cannot guarantee a unique identification of the rectangles in the merged node. However, if we partition B-regions, the grids of the two merged regions are part of the common grid, which follows from the properties of B-regions, see section 3. If V and W are B-partions of their B-regions B(V) and B(W) with $B(V) \cap B(W) = \emptyset$, then V and W are also B-partitions of $B(V \cup W)$. For a unique identification of the rectangles in a B-partition, it is necessary that the level of the z-values must be at least the same as the level of the B-partition.

Obviously, a shortcoming of a grid representation is that we do not maintain the minimal property of the rectangles in the directory. The rectangles only enclose the minimal rectangles. At first glance, we will expect more disk accesses for retrieval operations. However, we have gained a high fan out in the directory nodes. For example, let us assume 2-dimensional keys where each component requires 4 bytes. Then for an entry consisting of a rectangle and a pointer 4*4+2=18 bytes are necessary for the exact representation where 2 bytes are used for the pointer referring to the subtree. For a grid representation generally two bytes per z-value are sufficient. Therefore the fan out increases by the factor 18 : 6. This factor will be even better for higher dimensions. The improvement of the fan out is more important for the performance of the buddy-tree than giving up part of the minimal property.

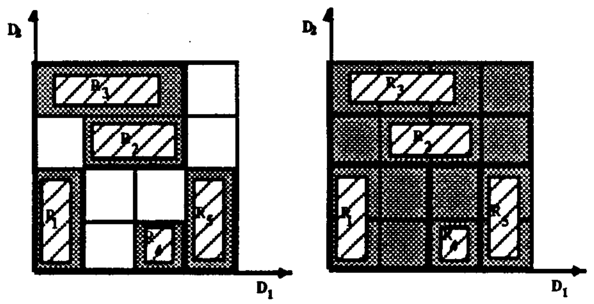


Fig. 4.1: Representation of the minimal rectangles $R_1,..,R_5$ in the buddy-tree (left side) and in the multi level grid file (right side)

Our approach for the organization of the directory is similar to that of the multi-level grid file [KW85]. One difference is that in case of the multi-level grid file (mlgf) only B-regions can be represented in the directory, whereas the representation of the buddy-tree is more exact by using two z-values, see figure 4.1. The buddy-tree comes closer to the minimal property than the mlgf.

5. Algorithms

In the following we describe the algorithms of the buddy-tree for an exact match query, range query and insertion. We decided to describe the algorithms in a programming language, in our case in Modula-2, because in our opinion it is more exact than to use pseudo language. For the sake of ease of understanding we have introduced some modifications to Modula-2. We have tried to use only some special type definitions and avoid the pointer concept at all. Therefore the description of types and procedures is quite different compared to the real implementation.

```
TYPE
Key  = ARRAY [1..d] OF KeyType;   (* KeyType is an
atomar type where the operation '>' is well defined. *)

DataRec = RECORD K: Key; info: ARRAY OF BYTE
END;     (* The info part of the type DataRec is not of
interest *)

Rectangle = RECORD low,high: Key END;
(* A rectangle is described by its lower left and right
upper corner. *)

DirEntry = RECORD R: Rectangle; next: FilePos END;
(* next is an address of a block stored on secondary
storage *)

BuddyNode = RECORD
                pos : SHORTCARD;
                CASE dir: BOOLEAN OF
                   TRUE:dirnode: ARRAY OF DirEntry
                   FALSE:datanode:ARRAY OF DataRec
                END
              END;
(* For a node in the tree pos is the number of actually
stored entries (dir = TRUE) or records. *)
```

First of all, we introduce the definitions of types. In our types *BuddyNode* and *DataRec* we have not specified the size of the arrays, because it will not be of interest for the description of the algorithms. In our algorithms we use Get-functions to return some derived information from a node. More exactly, *Get_Position* (node, {rectangle or key}, pos) returns the position (which is larger than the input parameter pos) of the first rectangle or data record in a node intersecting with the key or rectangle given as input. If no such position is found, 0 will be returned. The functions *Get_Entry* (node,pos), *Get_Record* (node, pos) and *Get_Node* (node, pos) provide for a given node and a position the corresponding entry, record and (son) node, respectively. Using these functions the algorithms *Emq* and *RQ* performing exact match and range query are completely described in the following.

```
PROCEDURE Emq (VAR node: BuddyNode; VAR
pos: SHORTCARD; K: Key);
(* returns the last node and the position of the touched
entry where the search finishes *)

 BEGIN
      pos := Get_Position(node, K, 0);
           (* pos = 0 <=> no entry found *)
       IF (pos # 0) AND node.dir THEN
           node := Get_Node(node, pos);
           Emq(node , pos, K)
      END;
 END Emq;

PROCEDURE  RQ(node: BuddyNode; R: Rectangle);
(* performs a range query, where answers are written on
output *)
VAR pos: SHORTCARD;

BEGIN
    pos := Get_Position(node, R, 0);
    WHILE pos # 0 DO
        IF node.dir THEN
            RQ(Get_Node(node, pos), R)
        ELSE
            (* Output the record returned by
            Get_Record(node,pos) *)
        END;
        pos := Get_Position(node, R, pos)
    END;
END RQ;
```

Let us mention that an exact match query is restricted to one path of the tree which can be easily seen in the algorithm *Emq*. Considering the algorithms, there is not much difference between the algorithms *Emq* and *RQ*. For the *Emq* algorithm we assume that at most one answer is allowed, whereas the *RQ* algorithm can obviously deliver a set of answers.

For the *Insert* algorithm more explanations are necessary. First of all, there are some functions requiring a node as input. The function *SecAddress* returns an address on secondary storage for the node and the function *Overflow* checks whether the node contains overflow records or entries. An entry or a record is inserted in a given node using the procedure Put_Entry or Put_Record, respectively. A procedure *Mergeable* asks a given node, whether entry (specified by its position) can be merged with an arbitrary other entry in the node. Finally, the procedure *Merge* executes a merge where the modified entry is stored at the position newpos (see algorithm *Insert*).

```
PROCEDURE Insert (VAR node: BuddyNode; drec:
DataRec);
VAR   newnode: BuddyNode;
      pos    : SHORTCARD;
      entry  : DirEntry;
BEGIN
    Emq(node, pos, drec.K);
    WHILE node.dir DO
        entry.R.low := drec.K; entry.R.up := drec.K;
        entry.next := SecAddress(newnode);
        Put_Entry(node, entry);
        (* insertion of an entry in a node *)
        newpos := node.pos;
        IF Mergeable(node, newpos) THEN
            Merge(node, newpos);
            node := Get_Node(node, newpos)
        ELSE
            IF Overflow(node) THEN Split(node) END;
            newnode.dir := FALSE; newnode.pos := 0;
            node := newnode
        END (* IF *)
    END (* WHILE *);
    IF pos # 0 THEN
        WriteErrorMsg("Record exists in the file");
    ELSE
        Put_Record(node,drec);
        (* insertion of a record in a node *)
        IF Overflow(node) THEN Split(node) END;
      END;
END Insert;
```

The most difficult case for an insertion appears, if the exact match query ends in a directory node. In this case, a new directory entry is created where the rectangle is described by the point which should be inserted. For this degenerated rectangle we search for a buddy where the corresponding node is not completely filled to accomodate an additional entry or rectangle (*Mergeable* = TRUE). Then we try to insert the record into this node, see figure 5.1. If no mergeable node can be found, a new data page is allocated where the record is inserted.

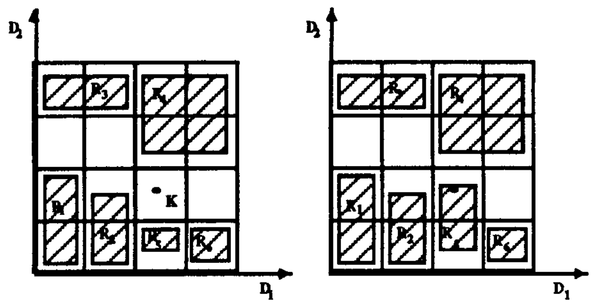


Fig 5.1: Solving the problem of an insertion where the record K falls into non-partitioned data space

In comparison to the multilevel grid file the buddy-tree avoids directory nodes with one entry and thus the buddy-tree is not balanced. However, we want to emphasize that this unbalanced directory reduces the cost for all operations in comparison to an artificially balanced directory! This is exactly the reason why the buddy-tree guarantees a linear growth of the directory in the number of records.

```
PROCEDURE Split (VAR node: BuddyNode);
VAR   fnode,newnode: BuddyNode;
      entry : DirEntry;
      axis  : [1..d];
      pos   : SHORTCARD;

BEGIN
    ComputeFather(fnode, node, pos);
    axis := Get_Splitaxis(node);
    DivideEntries(node, axis, newnode);
    entry.R := Get_MBB(node);
    entry.next := SecAddress(node);
    Update_Entry(entry,fnode,pos);
    entry.R := Get_MBB(newnode);
    entry.next := SecAddress(newnode);
    Put_Entry(fnode,entry);
    (* insertion of an entry in the node *)
    IF Overflow(fnode) THEN
        Split(fnode)
    ELSE
        Minimize(fnode)
    END;
END Split;
```

The insertion of a record is restricted to one path of the buddy-tree. This will be more clear by considering the split algorithms. Similar to B-tree algorithms a split can propagate up to the root, but cannot leave the top-down search path.

Since the split is the most complicated algorithm of the buddy-tree, we will go through the algorithm step by step. At first, the father node is evaluated for a given node named split node, by calling the procedure *ComputeFather*. Additionally, in the father node the position of the entry referring to the split node is computed. This procedure has to handle two exceptions. The first occurs if the split node is the root. Then a new root is created and filled with one entry referring to the split node. The second exception occurs, if the split node is a data node not stored on the deepest level of the buddy-tree. In this case a new father node is created with one entry referring to the original split node. Thus the level of the split node is incremented.

In the second step, the axis is determined in which the split should be executed. If we have several possibilities for the choice of a split axis, that one is selected where the margin of both resulting rectangles is the smallest. Then the procedure *DivideEntries* divides the records or entries into two groups corresponding to a hyperplane which is perpendicular to the split axis. More exactly, the hyperplane is determined by halfing the B-rectangle B(R) where R belongs to the entry in the father node which refers to the split node. One group of records or entries remains in the old node and the other group is stored in a new node. There is only the guarantee that the groups contain at least one entry or record. Similar to the grid file, the split strategy of the buddy-tree depends on the data space, but not on the stored data. Thus one of the advantages of the buddy-tree is that performance will be almost independent of the sequence of insertions.

The next four statements in the procedure Split describe the update of the old and the insertion of the new entry in the father node. The procedure *Get_MBB* computes the minimal bounding box of all rectangles in a given node. At last, the father node is checked for an overflow record and it is possibly split. In the other case the procedure Minimize is called which guarantees the minimal property of all possibly affected rectangles on the path. This is done at most once per level.

```
PROCEDURE Minimize (node: BuddyNode);
VAR   fnode : BuddyNode;
      entry : DirEntry;
      pos   : SHORTCARD;
BEGIN
    IF IS_Root(node) THEN RETURN END;
    ComputeFather(fnode, node, pos);
    entry := Get_Entry(fnode,pos);
    IF entry.R # Get_MBB(node) THEN
        entry.R := Get_MBB(node);
        Update_Entry(fnode,entry,pos);
        Minimize(fnode);
    END;
END Minimize;
```

We have not yet described one important feature of the split algorithm. If the distribution of the entries is rather uneven (e.g. one node is filled at most 30 %), then we will look for a buddy which can possibly be merged with the underfilled node.

In this paper, we do not present the deletion algorithm. But let us emphasize that, as it is true for insertion, deletion is restricted to one path of the buddy-tree.

6. Performance Comparison

In the following, we give a brief summary of our standardized testbed of PAMs described in detail in [KSSS89]. For justifying the choice of PAMs selected for our comparison we refer to the classification of multidimensional PAMs in table 1. Considering class C1, the most promising structures definitely are the interpolation-based grid file and the balanced multidimensional extendible hash tree. However, both structures can be obtained as a special case of the buddy-tree by restricting the properties of the regions. Therefore these two PAMs need not to be implemented. We do not include the best multidimensional dynamic hashing scheme without directory, PLOP hashing, since it is efficient only for weakly correlated data, but not for strongly correlated data. From class C 1 we selected the 2-level grid file because its efficient fine-tuned and well tested Modula-2 implementation by Klaus Hinrichs [Hin 85] is generally available which we thankfully acknowledge.

From class C 4 we omitted the B^+-tree storing z-values from our comparison. Instead, we decided to implement the BANG file and the hB-tree, because they are both improvements of the basic B^+-tree storing z-values. Obviously, we decided to implement the buddy-tree (class C 3) due to its non-complete partition of the data space which results in avoiding to partition empty data space. Since the concept of the twin grid file (class C 2) of organizing two dependent grid files at the same time is generally applicable to any PAM, we did not include it in our comparison. It might be worth investigating the application of this principle to the winners of our comparison. As a measuring stick we use our buddy-tree.

We ran our comparisons on SUN workstations (3/60) using Modula-2 implementations of the selected PAMs. We took seven different data files (F1) - (F7) of 2-dimensional records into account where due to space limitation the distributions of the data are described by the plotted points, see figure 6.1. The 7th data file (F7) contains uniformly distributed data. Each of the data files with the exception of one contains 100,000 records. The file (F6) belonging to the distribution RealData consists of 85,549 records of real cartography data representing the elevation lines in a "rolling-hill-type" area in the Sauerland, West Germany. The points are obtained as interpolation points of the elevation lines. Since the data is originally stored in a quad-tree, it is inserted in a sorted sequence which is due to the partitioning sequence of the quad-tree. We thankfully acknowledge receiving this data from the Landesvermessungsamt NRW, Bonn, West Germany. Let us emphasize that the Bit Distribution (F3) bit(z), $0 \leq z \leq 1$, was included with the choice of $z=0.15$, because it is the worst case distribution of the buddy-tree when z becomes small.

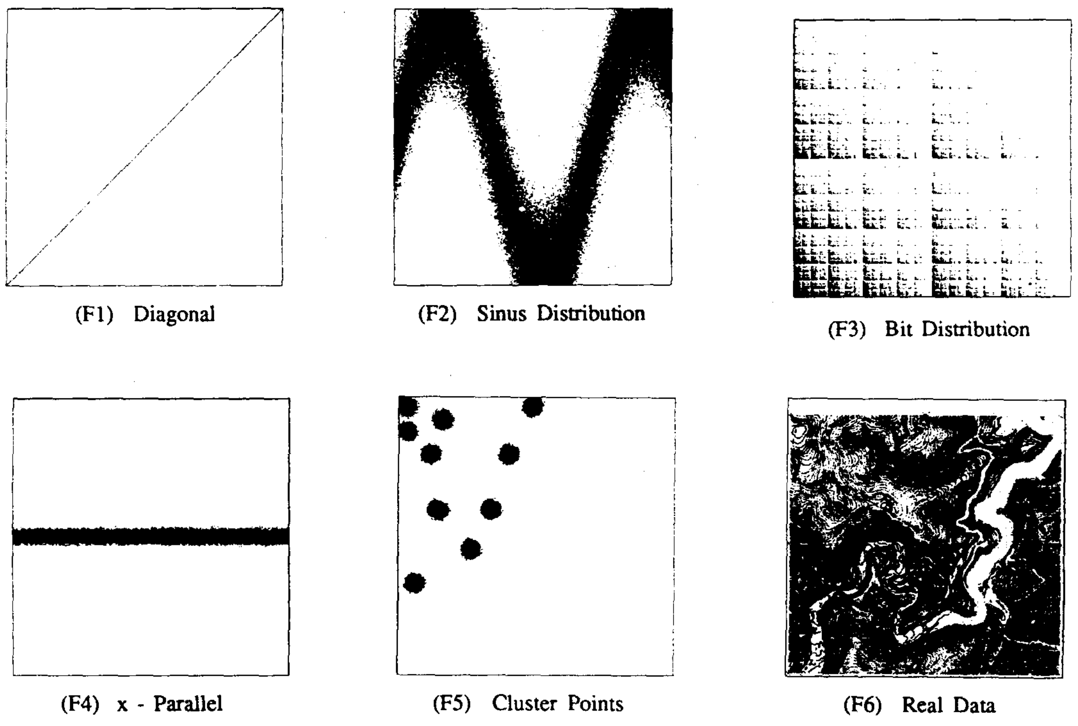


Fig. 6.1: Data distributions

To demonstrate the performance for range queries we generated five groups of 20 range queries. The regions of the first three groups are squares varying in size from 0,1 %, 1 % to 10 % relatively to the data space. The 4th and 5th group are partial match queries where the y- and x-value are unspecified, respectively. For all operations, we have measured the number of disk accesses per operation.

In table 6.2 for the parameters *stor* (average storage utilization) and *insert* (average cost for an insertion) we computed the unweighted average over all seven data files. As an indicator for the average query performance, we present the parameter query average which is averaged (unweighted) over all five query types for each distribution and then averaged over all seven distributions. The goal of this indicator is to help make things more clear, at first glance; however, we are aware that such an average implies a loss of information. The loss of information is considerably less in table 6.3 where the parameter query is displayed for each distribution as an average over all five types of queries. For the detailed description of all experiments and all results the interested reader is referred to [KSSS 89].

	query average	store	insert
hB-tree	164.1	68.6	2.80
BANG-file	131.9	67.9	2.49
grid file	148.5	58.3	2.56
buddy-tree	100.0	64.9	2.78

Table 6.2: unweighted average over all 7 distributions

	uniform	sinus	bit	x-par.	real data	diagonal	cluster
hB-tree	107.8	110.4	68.6	125.2	128.2	369.7	238.6
BANG-file	97.5	113.0	83.0	133.8	132.6	240.1	123.4
grid file	97.3	104.6	114.0	134.0	100.6	352.1	137.0
buddy-tree	100.0	100.0	100.0	100.0	100.0	100.0	100.0

Table 6.3: unweighted average over all 5 types of queries depending on the distribution

In order to keep the performance comparison manageable (we already had more than 2.7 million insertions), we have chosen the page size for data pages and directory pages to be 512 bytes which is at the lower end of realistic page sizes. Using small page sizes, we obtain similar performance results as for much larger file sizes, e.g. a doubling of the page size can accomodate an eight times higher file size within the same directory height for tree-based directories (BANG file, HB-tree, buddy-tree). We want to emphasize that the grid file implementation [Hin 85] always keeps the 1st level grid directory in main memory whereas for the other PAMs only the root page of the directory is main memory resident. Since it was crucial to change the grid file implementation to allowing only one root page of the directory in main memory, we accepted that the relative ranking of thegrid file is too good in comparison to the other structures. To clarify this: for the Diagonal Distribution the 1st level grid directory needed 45 directory pages in main memory, which is sufficient for BANG file and buddy-tree to keep the complete directory in main memory. Thus the rating of the grid file in a comparable environment would be considerably worse. Due to the main memory-resident directory, increasing page size implies that the relative performance of the grid file will decrease in comparison to the other structures.

Considering table 6.2 the buddy-tree offers itself to be the winner of our comparison. It is interesting to observe that the buddy-tree does not fulfill the often cited rule "best storage utilization - best query performance". Let us take a closer look at the different distributions in table 6.3. The only distributions where the buddy-tree is not the winner are the Uniform Distribution and the Bit Distributions. As mentioned before the Bit Distribution is the worst case distribution for the buddy-tree. Even for its worst case distribution the buddy-tree is better than the grid file. This underlines the robustness of our structure. For the Uniform Distribution the buddy-tree is within a 3% margin of the grid file, the winner. This is surprising for a scheme designed for nonuniform data incorporating the complex structural concept of not partitioning empty data space. In all distributions, with the exception of the Uniform and Bit Distribution, the buddy-tree is the clear winner in the average query performance. Summarizing we can state that the buddy tree clearly outperforms its competitors if at least one of the following two data characteristics occur: (C1) densely populated and unpopulated areas vary over the data space, (C2) sorted data is inserted. Sorted insertions frequently occur in real-life applications, either sorted by some local ordering such as clusters or quadrants or by lexicographical ordering. First results in a performance comparison with rectangles underlign the superiority of the buddy-tree.

7. Conclusions and Future Work

In this paper, we proposed the buddy-tree, a new dynamic multidimensional access method. Contrary to previously suggested point access methods, the buddy-tree generates the rectangular regions in its directory as minimal as possible. Therefore, the data space is not completely covered by these regions. In particular empty data space is not reflected in the directory. Moreover, the buddy-tree avoids overlap in the directory nodes using a generalization of the so-called buddy-system. Additionally, we propose a general implementation technique for the directory increasing the fan out of the directory nodes. Using our standardized testbed, we present a performance comparison of the buddy-tree with other access methods demonstrating the superiority and robustness of the buddy-tree.

Our current and future work in the area focuses in the following tasks:

- We examine whether a controlled overlap in the directory (e.g. the twin technique [HSW88]) can improve storage utilization
- A pack algorithm is integrated in our implementation avoiding underfilled data nodes
- Different techniques to generate SAMs based on the buddy-tree will be implemented and investigated.

References:

[BKSS90] Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: 'The R*-tree: An efficient and robust access method for points and rectangles', in Proc. ACM SIGMOD International Conference on Management of Data , May 23-25, 1990, Atlantic City, USA, 322-331, 1990

[BM72] Bayer,R., McCreight, E.: Organization and maintenance of large ordered indexes, Acta Informatica 1,3 173-189,1972

[Bur83] W.A. Burkhard: 'Interpolation-based index maintenance', BIT 23, 274-294, 1983

[FSR87] C. Faloutsos, T. Sellis, N.Roussopoulos: 'Analysis of object oriented spatial access methods, Proc. ACM SIGMOD Int.Conf. on Management of Data,426-439, 1987

[Fre87] M. Freeston: 'The BANG file: a new kind of grid file', Proc. ACM SIGMOD Int. Conf. on Management of Data, 260-269, 1987

[Gut84] A. Guttman: 'R-trees: a dynamic index structure for spatial searching', Proc. ACM SIGMOD Int. Conf. on Management of Data, 47-57, 1984

[HCKW90] E. Hanson, M. Chaabouni, C.-H. Kim, Y.-W. Wang:'A predicate matching algorithm for database rule systems', ACM SIGMOD 90, 271-280, 1990

[Hin85] K.Hinrichs:'The grid file system:implementation and case studies for applications', Dissertation No.7734, Eidgenössische Technische Hochschule(ETH), Zuerich, 1985

[HSW88] A. Hutflesz, H.-W. Six, P. Widmayer: 'Twin grid files:space optimizing access schemes',Proc.ACM SIGMOD Int. Conf. on Management of Data, 183-190, 1988

[Kri84] H.P. Kriegel: 'Performance comparison of index structures for multikey retrieval', Proc. ACM SIGMOD Int. Conf. on Management of Data, 186-196, 1984

[KS86] H.P. Kriegel, B. Seeger: 'Multidimensional order preserving linear hashing with partial expansions', Proc. Int. Conf. on Database Theory, Lecture Notes in Computer Science 243, 203-220, 1986

[KS88] H.P. Kriegel, B. Seeger: 'PLOP-Hashing: a grid file without directory', Proc. 4th Int. Conf. on Data Engineering, 369-376, 1988

[KS89] H.P. Kriegel, B. Seeger: 'Multidimensional quantile hashing is very efficient for non-uniform distributions', in Information Sciences 48, 99-117, 1989

[KSSS89] H.P. Kriegel, M. Schiwietz, R. Schneider, B. Seeger:Performance comparison of point and spatial access methods,Proc.Symp. on the Design and Implementation of Large Spatial Databases, Santa Barbara, July 17-18,1989. Lecture Notes in Computer Science 409, 89-114, 1989.

[LS89] D.B. Lomet, B. Salzberg: The hB-tree: A robust multiattribute search structure, in Proc. of the Fifth Int. Conf. on Data Engineering, Feb. 6-10, 1989, LosAngeles, also available as Technical Report TR-87-05, School of Information Technology, Wang Institute of Graduate Studies.

[NHS84] J. Nievergelt, H. Hinterberger, K.C. Sevcik: The grid file: an adaptable, symmetric multikey file structure', ACM Trans. on Database Systems, Vol. 9, 1, 38-71, 1984

[Ore82] J.A. Orenstein:'Multidimensional tries used for associative searching', Inf. Proc. Letters 14, 4, 1982, 150-157

[OM84] J.A. Orenstein, T.H. Merrett: 'A class of data structures for associative searching', Proc 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 181-190, 1984

[Oto84] E. J. Otoo: 'A mapping function for the directory of a multidimensional extendible hashing', Proc. 10th Int. Conf. on Very Large Databases, 491-506, 1984

[Oto86] E. J. Otoo, : 'Balanced multidimensional extendible hash tree', Proc. 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 110-113, 1986

[Ouk85] M. Ouksel: 'The interpolation based grid file', Proc. 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1985

[Rob81] J. T. Robinson: 'The K-D-B-tree: a search structure for large multidimensional dynamic indexes', Proc. ACM SIGMOD Int. Conf. on Management of Data, 10-18, 1981

[See89] B. Seeger.: 'Design and implementation of multidimensional access methods' (in German), PhD thesis, Department of Computer Science, University of Bremen.

[SK88] B. Seeger, H. P. Kriegel: 'Design and implementation of spatial access methods', Proc. 14th Int. Conf.on Very Large Databases, 360-371, 1988

[Tam82] M. Tamminen: 'The extendible cell method for closest point problems', BIT 22, 27-41, 1982

[WK85] K.-Y. Whang, R. Krishnamurthy: 'Multilevel grid files', Technical Report, IBM Research Lab., Yorktown Heights, 1985