The Binary-Radix Bucket-Region-Directory: A Simple New Directory for the Grid File

Ulrich Finke, Klaus Hinrichs, Ludger Becker
FB 15 - Informatik, Westfälische Wilhelms - Universität,
D - 4400 Münster, Germany

Abstract

The grid file introduced by Nievergelt et al. in [1] is a dynamic, symmetric multikey file structure. It organizes highly dynamic sets of multidimensional data on secondary storage in such a way that different types of queries can be performed using few disk accesses. We present the new BR^2-directory structure for the grid file. The BR^2-directory is implemented by a tree. It grows linearly with the number of data buckets and reduces both the CPU-costs for directory operations and the I/O costs for directory pages.

Keywords: File structures, GridFile, Directory structures, Buddy system, Multikey searching, Splitting Buckets, Merging Buckets, Dynamic storage allocation

1 Introduction

The grid file partitions the k-dimensional data space $D=d_1 \times d_2 \times \ldots \times d_k$ by an orthogonal grid. d_i $(1 \le i \le k)$ denotes the domain underlying dimension i. The grid is defined by (k-1)-dimensional hyperplanes. Each hyperplane is determined by a data value in one dimension, called a boundary. The boundaries of each dimension are represented by a scale. Two successive boundaries in a scale determine a binary radix interval. A new boundary is generated by halving such a binary radix interval (Fig. 1).

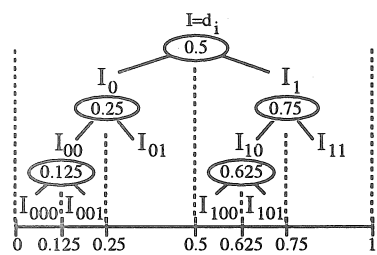


Fig. 1

Each cell of the grid is assigned to a physical disk block on secondary storage, called a bucket. All data points which lie in a grid cell can be found in the disk block assigned to this grid cell. A directory structure represents and manages the assignments of grid cells to buckets. The mapping is surjective: several grid cells may share a bucket. The grid cells covered by a bucket form the bucket's region (Fig. 2). The mapping has to adhere to the following rules which determine the buddy system:

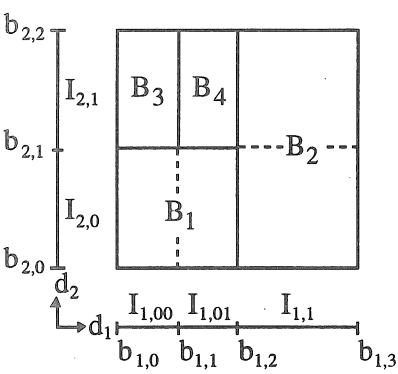


Fig. 2

- The bucket region must be a k-dimensional rectangular box:
  $B_n = I_1 \times I_2 \times \ldots \times I_k$, with $\forall_{j \in \{1 \ldots k\}}\ I_j \subseteq d_j$.
- Each interval I_j is a binary-radix interval.

Insertions and deletions of data points modify the grid dynamically. Insertions may lead to splitting of

overflowing buckets. If a bucket region consists of two or more grid cells it can be split along a dividing hyperplane. Otherwise a new boundary must be created by halving a bucket region interval I_j. The bucket-region-condition

$$\max\left(\frac{|d_1|}{|I_1|},\frac{|d_2|}{|I_2|},\ldots,\frac{|d_k|}{|I_k|}\right) \leq 2 \min\left(\frac{|d_1|}{|I_1|},\frac{|d_2|}{|I_2|},\ldots,\frac{|d_k|}{|I_k|}\right)$$

ensures that almost "quadratic" bucket regions are generated (heuristic). $|d_j|$ denotes the "length" of domain d_j, $|I_j|$ denotes the length of interval I_j.

Deletions of data points may lead to merging of underflowing buckets. After merging two buckets it may become necessary to remove a boundary, i.e. a row of the grid, which is no longer used. Merging is not simply the reversal of the split process. The buddy system may allow an underflowing bucket to merge with one of k potential candidates.

[1] left open some important questions such as the implementation of the scales and the grid directory. An obvious approach is the *grid array* [2], which represents the grid by a unique multidimensional dynamic array. Each grid cell corresponds to an array element. However, inserting and removing boundaries, i.e. inserting or removing a row in any dimension, are time consuming operations since virtually all array elements have to be moved. Furthermore the grid array may grow superlinearly with the number of buckets for non uniform data distributions [3].

The *region representation* [4] stores all bucket regions together with the bucket addresses in a linear list. This approach guarantees that the directory grows linearly with the number of buckets. Both the explicit representation of bucket regions and the list implementation lead to time consuming search- and management-operations.

The objective of our work was to develop a new kind of directory which guarantees a linear growth with the number of buckets, ought to be easy to implement and supports efficient directory management operations.

2 The BR^2-directory

Each boundary of a scale divides the data space into two parts. The BR^2-directory represents the subdivision of the data space by a tree-like structure.

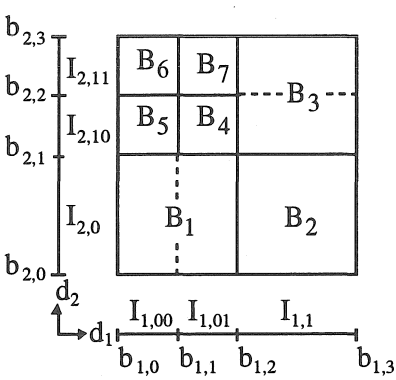


Fig. 3

2.1 Construction of the BR^2-tree

We explain the construction of a BR^2-directory for the two-dimensional partitioning of the data space given in Fig. 3. Two scales each containing four boundaries $b_{j,0} \ldots b_{j,3}$ determine the grid consisting of nine cells. The binary sequence S determines how an interval $I_{j,S}$ is obtained by subsequent bisection from d_j: A zero (0) means selection of the lower interval, a one (1) selection of the upper interval. $B_1 \ldots B_7$ represent the buckets. Except for B_1 and B_3, they only cover one grid cell.

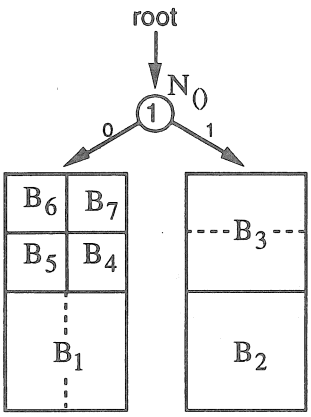


Fig. 4

Each boundary partitions the data space into two. For a grid file which has been generated by insertions only the buddy system guarantees the existence of at least one boundary which does not intersect any bucket region, e.g. $b_{1,2}$ and $b_{2,1}$ in Fig. 3. We select boundary $b_{1,2}$ which separates the buckets B_1, B_4...B_7 from the buckets B_2 and B_3 (Fig. 4). Therefore the root node N_0 of the BR^2-tree obtains an entry for the dimension of $b_{1,2}$, i.e. node dimension $d_{N_0} = 1$.

We continue by selecting boundaries and represent them by new nodes, which contain their dimension as an entry. Fig. 5 shows one of the possible BR^2-trees for our example. The pointers to the leaves are the identifiers of the buckets on disk.

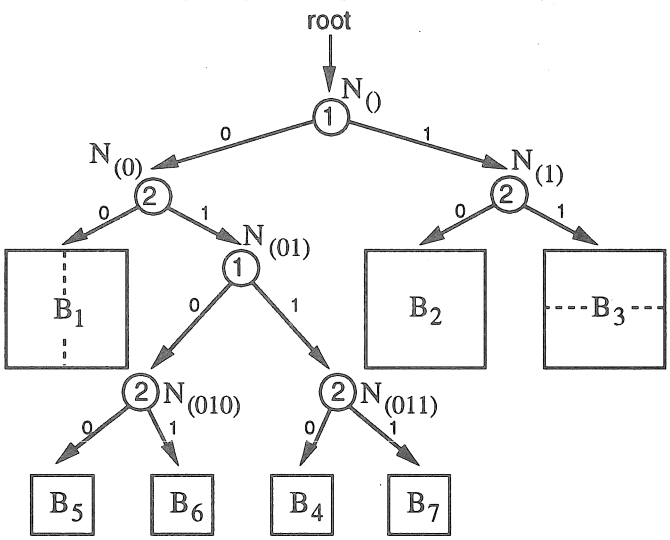


Fig. 5

2.2 Properties of BR^2-trees

Let n be the number of buckets. We summarize some properties of BR^2-trees:

- In binary trees the number of internal nodes is equal to the number of leaves minus one. Therefore the size of the BR^2-tree grows linearly in n: $\Theta_{space}(n)$.
- Each internal node corresponds to an unique boundary in one of the scales. A boundary may be assigned to more than one internal node.
- Each node N (internal node or leaf) represents a rectangular region D_N of the data space; this region is the Cartesian product of binary-radix intervals represented by the scales. The depth of a node or leaf determines the size of this region.

- Each internal node or leaf owns a unique path, which can be described by a bit-sequence S. In Fig. 5 these bit sequences S form the node identifiers $N_{(S)}$. If S_1 is the bit-sequence of a node N_1 and S_2 the bit-sequence of a node or leaf N_2, then:
  S_1 is prefix of $S_2 \Leftrightarrow D_{N_1} \supseteq D_{N_2}$.
- The BR^2-representation of the partitioning is ambiguous. In most cases there exist several different equivalent BR^2-trees, e.g. four trees for the example of Fig. 3.

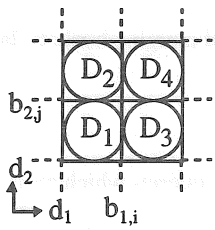


Fig. 6

Each tree can be obtained from any equivalent tree by performing a number of transformations. Fig. 6 shows a subdivision of one part of the data space. $D_1 \ldots D_4$ could be bucket regions or further be subdivided. The two equivalent BR^2-subtrees representing this subdivision are shown in Fig. 7.

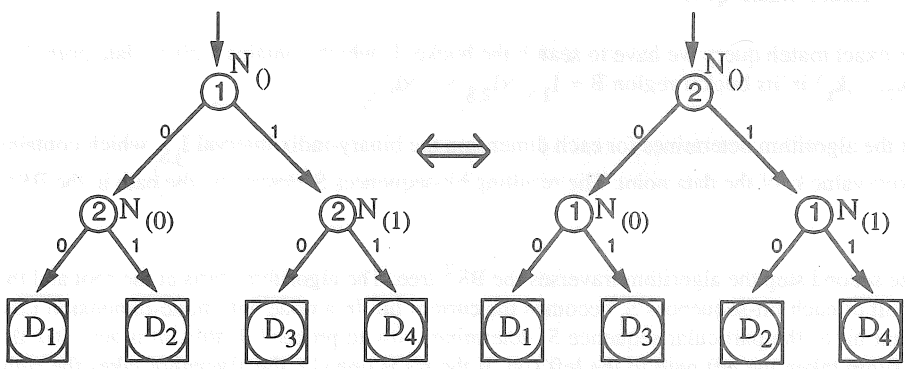


Fig. 7

Each subtree can be obtained from the other subtree by exchanging the node-dimension of the node N_0 with the node-dimension of sons $N_{(0)}$ and $N_{(1)}$ and exchanging the references of D_2 and D_3. Such a d-transformation can always be carried out if the two direct sons of a node have the same node-dimension which must be different from the node dimension of the father.

The dimensions which can appear in a node N after any d-transformations have been performed in the subtree rooted at N are called $PND_N \subseteq \{1,\ldots,k\}$, the set of possible node dimensions of N. PND_N can be determined recursively:

- $PND_{Leaf} = \varnothing$.
- If $N_{(S)}$ is an internal node with node dimension $d_{N_{(S)}}$ and direct sons $N_{(S0)}$ and $N_{(S1)}$, then
  $PND_{N_{(S)}} = (PND_{N_{(S0)}} \cap PND_{N_{(S1)}}) \cup \{ d_{N_{(S)}} \}$.

For a node N the set PND_N does not change when any d-transformations are performed in the subtree rooted at N. The complexity to determine PND_N for all nodes N of a given BR^2-tree is: $\Theta_{Time}(n)$.

In an implementation of the BR^2-tree each node can be represented by 8 bytes; three bytes for each pointer to the sons and two bytes for additional information, e.g. the node dimension. Since the identifiers for buckets are stored in their father nodes we need $8(n-1)$ bytes for n buckets. This is less than is needed in the grid array representation (even under uniform data distribution) and the region representation.

2.3 Operations on BR^2-trees

Fundamental operations on directory structures are searching, splitting and merging buckets. In the following we discuss these operations.

2.3.1 Searching buckets

Directory representations have to support efficient exact match and range queries which return references to buckets containing the records to be searched for.

a) Exact match query

In an exact match query we have to search the bucket B which contains a given data point $P = (k_1,k_2,\ldots,k_k)$ in its bucket region $B = I_{1,S_1} \times I_{2,S_2} \times \ldots \times I_{k,S_k}$.

First the algorithm determines for each dimension the binary-radix interval I_{j,S_j} which contains the key-value k_j of the data point. The resulting bit-sequences S_j determine the path in the BR^2-tree.

In the second step the algorithm traverses the BR^2-tree: The algorithm starts at the root and the first bit of each bit-sequences S_j becomes the current bit. In a node with node-dimension j the current bit of the particular sequence S_j determines how to proceed. If this bit is zero (0), the algorithm takes the left path to the left son, if the bit is one (1), the algorithm takes the right path. Then the next bit from sequence S_j becomes the current bit. The algorithm terminates as soon as a leaf is reached.

The costs for this search in the BR^2-directory and the scales are for uniform data distribution $\Theta_{Time}(\log n)$.

b) Range query

The range query is formally defined as follows: Search all buckets B whose bucket regions B intersect a given query range $R = (I_1, \ldots, I_k)$ where I_j is an interval in the range of dimension d_j. The query is again performed in two steps:

- Determine for each interval $I_j = [l_j, u_j]$ two bit sequences L_j and U_j for the lower and upper bound of I_j in the binary-radix tree for dimension d_j.
- Traverse the BR^2-directory.

Remember each node N of the BR^2-directory is represented by an unique bit sequence S_N. For example the representation (or path) to bucket B_4 in Fig. 5 is $S_{B_4} = 1_0 \cdot 2_1 \cdot 1_1 \cdot 2_0$. The corresponding part of the data space is the cartesian product of k binary radix intervals which can again be described by k unique bit sequences. Let $S_{N,j}$ be the bit sequence for dimension j (e.g. $S_{B_4,2}=2_1 \cdot 2_0$).

A node N is relevant for the range query if all sequences $S_{N,j}$ $(1 \le j \le k)$ satisfy $L_j \le_s S_{N,j} \le_s U_j$ where $\le_s$ is the lexicographic order on bit strings. Starting at the root of the tree, we distinguish the following cases at a node N which has node dimension j (Fig. 8):

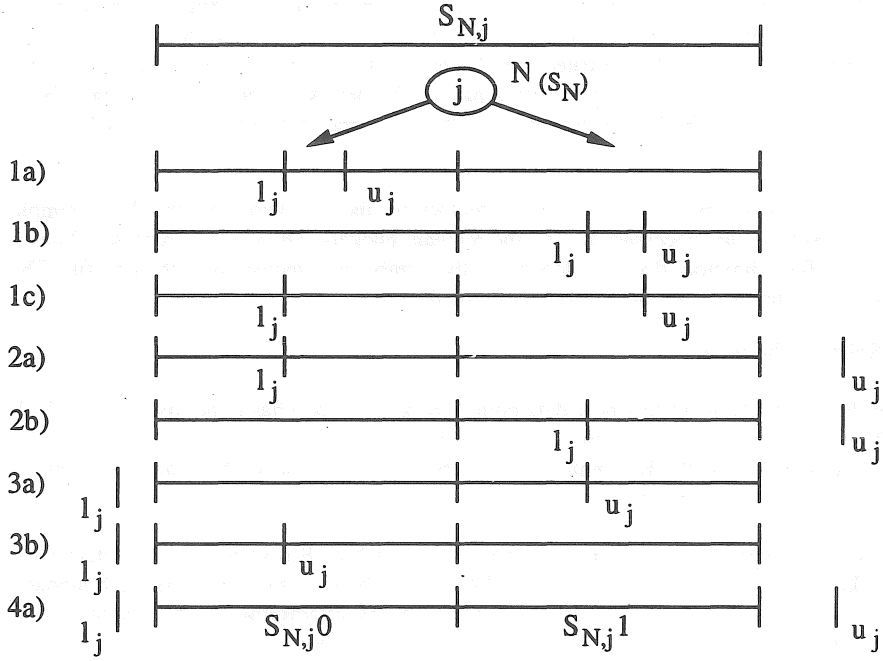


Fig. 8

1) $S_{N,j}$ is prefix of L_j and U_j, i.e. the binary radix interval corresponding to the region of node N in dimension j contains both the lower bound l_j and the upper bound u_j. If the current bits of the sequences L_j and U_j are both 0, we have to continue with the left son (1a), if the current bits are both 1, we have to continue with the right son (1b). If the current bits are different we have to visit both sons (1c).

2) $S_{N,j}$ is prefix of L_j, but not of U_j, i.e. the binary radix interval corresponding to the region of node N in dimension j contains the lower bound l_j. If the current bit of sequence L_j is 0 we have to visit both sons of N (2a). Otherwise only the right son is considered (2b).

3) $S_{N,j}$ is prefix of U_j, but not of L_j, i.e. the binary radix interval corresponding to the region of node N in dimension j contains the upper bound u_j. If the current bit of sequence U_j is 1 we have to visit both sons of N (3a). Otherwise only the left son is considered (3b).

4) $S_{N,j}$ is neither prefix of L_j nor of U_j, i.e. the binary radix interval corresponding to the region of node N in dimension j contains neither the lower bound l_j nor the upper bound u_j. Therefore we have to visit both sons (4a).

It is not necessary to construct the bit sequences of a visited node because the prefix properties can be inherited from the father. It is sufficient to introduce 2k status variables to describe the prefix properties of each bit sequence. The other tests in the algorithm are simple bit comparisons.

The BR^2-directory has the advantage that every bucket in the query range is returned exactly once. In the grid array representation a bucket may cover more than one grid cell and hence it may be returned more than once during range queries. Therefore we have to decide for each bucket whether it has already been visited. For this test we must construct its bucket region, and determine whether this region extends into a part of the data space which has already been processed. Constructing the bucket region from the grid array representation is a CPU-time consuming task.

The BR^2-directory is superior to the region directory for the following reasons. More simple and less comparisons are necessary to find the relevant buckets. The space to represent bucket regions in the BR^2-directory does not depend on the number of dimensions of the grid file. The overall space requirements are higher for the region directory.

2.3.2 Splitting buckets

A bucket B can overflow when a new data point $P = (k_1,k_2,\ldots,k_k)$ has to be inserted into B. If B overflows it has to be split and new disk blocks have to be allocated. We assume that the bit sequences S_j determining the BR^2-path to B have been computed as in the exact match query. Then the split can be performed as follows:

First the split dimension, i.e. the dimension according to which B is split, must be determined. If the bucket region covers more than one grid cell B can be split along an existing boundary. The existence of such a boundary can be determined by examining the bit sequences S'_j remaining after the traversal of the BR^2-tree. B can be split according to dimension j if S'_j is not the empty.

If no such dimension exists no existing boundary divides B's region. In this case we select a split dimension j and a boundary b_j as has been explained in section 1 (bucket-region-condition). We insert the boundary into the particular radix tree of scale j. In any case we can now split B by allocating a new internal node N and a new bucket B'. The old father of B now points to N. The node dimension of N is j, the left son is B, the right son is B'. All data points stored in B are distributed among B and B' depending on their key value in dimension j.

The insertion of a new boundary in a scale costs $O_{Time}(\log n)$ for uniform data distribution. The remaining costs of a split are $\Theta_{Time}(1)$.

2.3.3 Merging buckets

The grid file has to adapt gracefully to a varying distribution of records. Deletions of data points may lead to almost empty data buckets. In order to achieve a high storage utilization buckets may be merged if possible. The buddy system may allow an underflowing bucket to merge with one of k potential candidates. In a grid file of three or more dimensions merging of buckets according to the buddy system may lead to deadlocks (Fig. 9), i.e. buckets are generated which prevent each other from further merging since their resulting bucket region would not be rectangular.

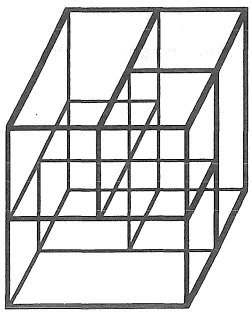


Fig. 9

Both the grid array and the region representation can represent such a deadlock configuration, but there exists no BR^2-tree for such a space partitioning. In general each space partitioning represented by a BR^2-tree is deadlock-free: In a BR^2-tree there exists at least one internal node which has two leaves as direct sons. Thus in each BR^2-directory we can find at least two buckets which can be merged if they become almost empty.

Some necessary criteria for finding a merge candidate B_2 for an underflowing bucket B_1 are:

(1) B_2 must have the same depth in the BR^2-tree.

(2) B_1 and B_2 can be merged only if their joint occupancy is well below the bucket capacity.

(3) If we have found a candidate B_2 according to (1) and (2), then B_1 and B_2 must have a common father node FATHER. We describe the path to a node N (internal node or leaf) by a sequence S_N which contains for each path node its dimension and the direction of traversal (see 2.3.1.b for an example).

Furthermore we denote the path from node N_1 to N_2 with $S_{N_1 \to N_2}$. Then the following holds for S_{B_1} and S_{B_2} (Fig. 10):

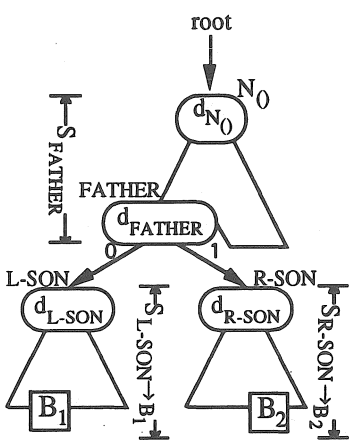


Fig. 10

$$S_{B_1} = S_{FATHER} \cdot (d_{FATHER})_0 \cdot S_{L\text{-}SON \to B_1}$$

$$S_{B_2} = S_{FATHER} \cdot (d_{FATHER})_1 \cdot S_{R\text{-}SON \to B_2}$$

The path branches out at the common father node. Since for a merge the buckets must be neighbors in data space, the paths $S_{L\text{-}SON \to B_1}$ and $S_{R\text{-}SON \to B_2}$ must satisfy the following conditions:

- The paths are not allowed to contain the father's node dimension d_{FATHER}.
- Let $S_{N,j}$ denote the path in dimension j; then the following condition must be true:

$$\forall_{j \in \{1 \ldots k\}} S_{L\text{-}SON \to B_1, j} = S_{R\text{-}SON \to B_2, j}$$

(4) $|PND_{FATHER}| \geq 2.$

If a bucket meets these conditions a sufficient check can be made:

(M1) [Select dimension]
Select a dimension j from PND_{FATHER}.

(M2) [Transform subtree]
Transform the subtree rooted at FATHER with d-transformations, such that j becomes the new node dimension of FATHER. The buckets B_1 and B_2 are descendants of their new father node L-SON or R-SON.

(M3) [Look for new father]
Make the particular son to the new father-node:
Either L-SON → FATHER or R-SON → FATHER.

(M4) [Merge buckets]
If B_1 and B_2 are direct sons of the new father, then merge the buckets and terminate.

(M5) [Loop or termination]
Determine PND_{FATHER}. If PND_{FATHER} satisfies condition (4) then continue with (M1), else terminate.

If the algorithm terminates in (M4), the buckets can be merged by copying all data points from B_2 to B_1 and replacing the father-node by B_1. If possible a search for other canditates follows a termination in (M5).

The total costs for finding all merge candidates of a given bucket are $O_{Time}(2^k)$ for uniform data distribution.

3 Implementation issues

We have presented the basic idea of the BR^2-directory representation. To improve performance we introduce some additional concepts:

2-level directory

Since in general a directory representation does not fit in main memory Hinrichs [2] proposed the 2-level directory (Fig. 11). The first level is the root directory, a scaled down version of the grid directory. The boundaries which determine the grid corresponding to the root (and page) directory(ies) are contained in the scales. An element of the root directory is a pointer to a directory page on secondary storage which contains the corresponding part of the grid directory. The elements of the subdirectories are references to data buckets. The constraints defined for the grid directory apply to the root directory as well as the subdirectories. In [2] the root directory and the subdirectories are represented by grid arrays. It is obvious that the 2-level technique can also be applied to the BR^2-directory representation.

Empty buckets

Since for non uniform data distributions the grid file can contain many empty buckets, the storage utilization may degenerate if all these empty buckets are allocated on disk. This degeneration is avoided by replacing each reference to a bucket B which becomes empty by a

special 'empty bucket'-reference - similar to a NIL-pointer in dynamic data structures. The bucket B can now be deallocated.

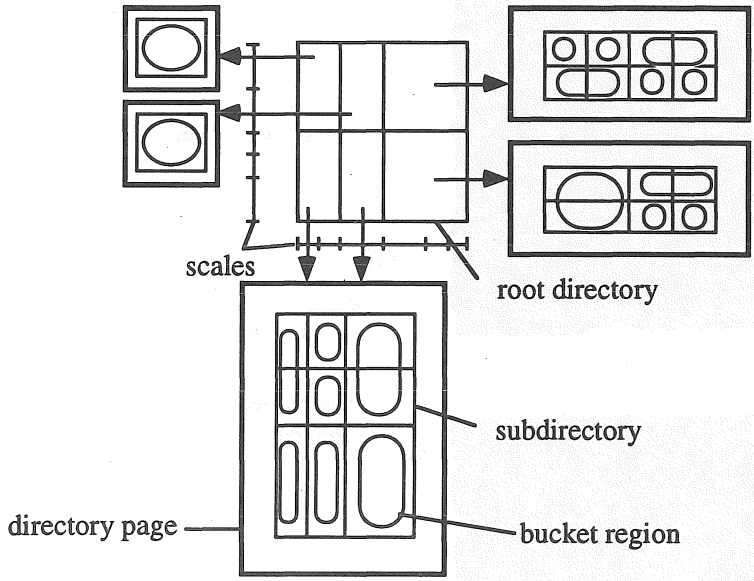


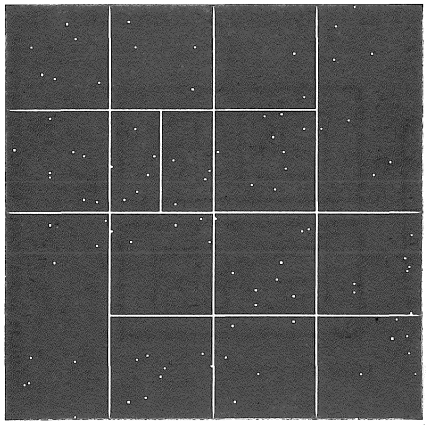
Fig. 11

Buffer management

The performance of operations on access methods is increased by using a part of the available main memory as a buffer. For a 2-level directory this buffer should contain the root directory and the scales, because every directory operation must access these parts. The remaining memory can be used for storing pages and buckets; the distribution of this part of the system buffer should be managed by a heuristic strategy like LRU. However, efficient operations on access methods require that a set of pages and buckets can be locked by operations, i.e. they cannot be removed from the system buffer.

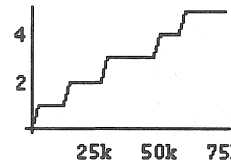
4 Tests

For tests we have implemented a BR^2-directory based grid file together with a test system called 'InsideGrid' which can be used to generate test data and to study the behavior of the directory. In the following we show the growth of the directory in the data pages (512 byte) for different data distributions. To get interesting results the page and bucket size is one data page and the length of a record is 50 bytes, so every bucket can store up to 9 records. Better results can be expected for larger pages and buckets. During every test 75000 records are inserted in a GridFile. The first, second and third diagram show the number of data pages used for the root directory, the directory pages and buckets. The number of empty bucket references (not allocated buckets!) is shown in the fourth diagram. For two dimensional data distributions an example of the corresponding partitioning of the data space is shown.

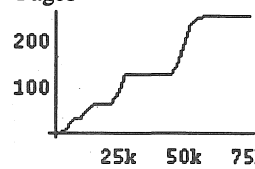


I. **Uniform data distribution:**

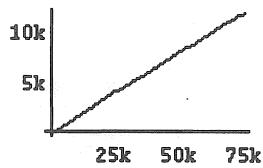
Root



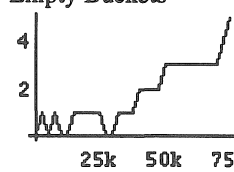
Pages

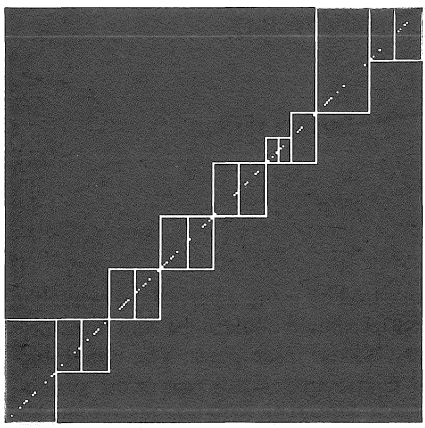


Buckets



Empty Buckets

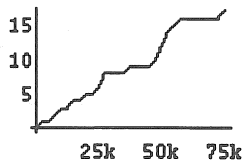




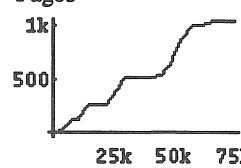
II. **Correlated data distribution:**

a. Diagonal through data space

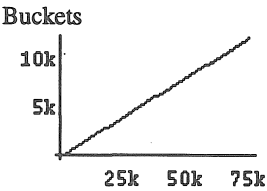
Root



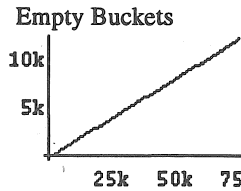
Pages

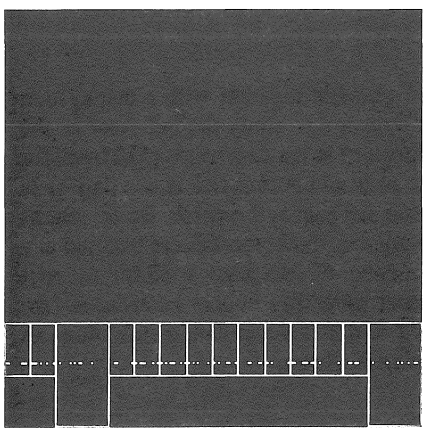


Buckets



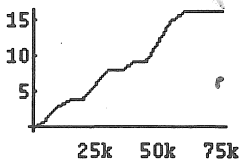
Empty Buckets



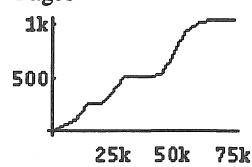


b. Second key is constant

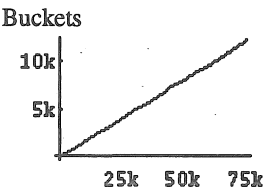
Root



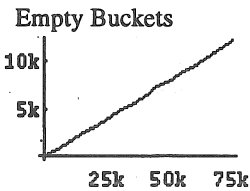
Pages

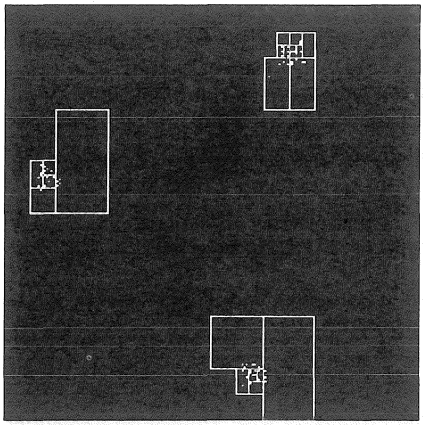


Buckets

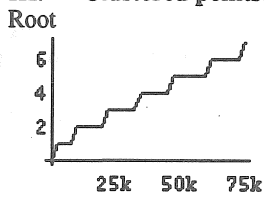


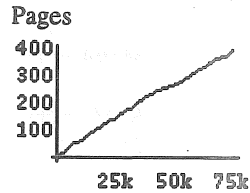
Empty Buckets

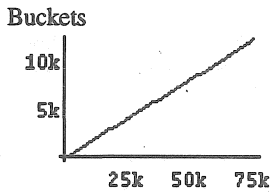


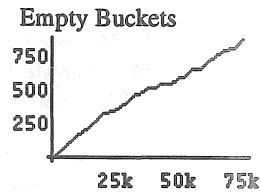


III. Clustered points:



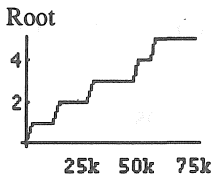


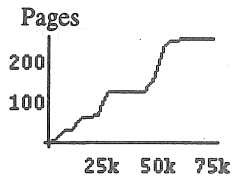


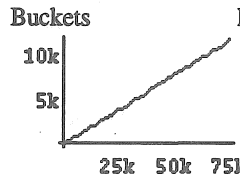


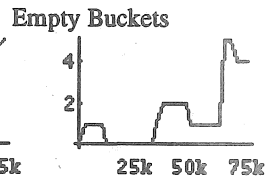
IV. **Data distribution by storing extended objects: (4-dimensional)**

a. Center transformation:

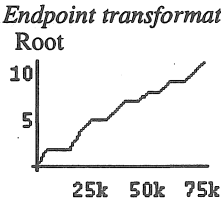


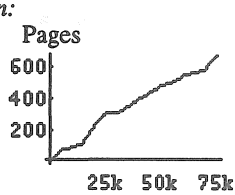


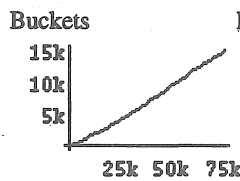


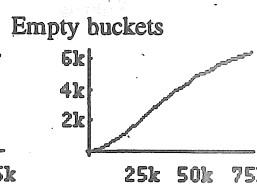


b. Endpoint transformation:









5 Conclusions

In this paper we have presented the BR^2-representation for the directory of a grid file. This representation guarantees that the directory grows linearly with the number of data buckets. Besides the algorithms for exact match and range queries we have developed evaluation strategies for other queries using grid files with a BR^2-representation of the directory, e.g. region and join queries [5], [6].

Since all elements of a BR^2-directory are simple binary trees, it is straightforward to apply techniques developed for concurrent operations on B-trees and other dynamic data structures [7], [8], [9] to a grid file with BR^2-directory.

References

[1] J.Nievergelt, H. Hinterberger, K.C. Sevcik: The Grid File: An Adaptable, Symmetric Multikey File Structure, ACM TODS 9, 1 (1984), 38 - 71.

[2] K. Hinrichs: Implementation of the Grid File: Design, Concepts and Experience, BIT 25 (1985), 569 - 592.

[3] M. Regnier: Analysis of Grid File Algorithms, BIT 25 (1985), 335 - 357.

[4] H. Hinterberger: Data Density: A Powerful Abstraction to Manage and Analyze Multivariate Data, Dissertation No. 8330, Eidgenössische Technische Hochschule (ETH), Zürich, 1987.

[5] L. Becker, U. Finke, K. Hinrichs: Efficient Join Processing with Multidimensional File Structures, Informatik Bericht Nr. 90 - 03, Universität-GH-Siegen.

[6] L. Becker, K. Hinrichs, U. Finke: A Cost Model for Query Processing with Grid Files, Informatik Bericht Nr. 91 - 01, Universität-GH-Siegen.

[7] C.S. Ellis: Concurrency in extendible hashing, Information Systems (1988), Vol. 13, No. 1, 97-109.

[8] Y. Kwong, D. Wood: Method for concurrency in B-trees, IEEE-TOSE (1982), Vol. 8, No. 3, 211-223.

[9] Shasha D., Goodman N.: Concurrent search structure algorithms, ACM-TODS (1988), Vol. 13, No. 1, 53-90