Balanced Multidimensional Extendible Hash Tree

Ekow J. Otoo
School of Computer Science
Carleton University
Ottawa, Canada, K1S 5B6.

Abstract

We present a method for designing a multidimensional order preserving extendible hashing scheme that allows the directory to grow almost linearly with the number of insertions, irrespective of the key distribution. Such robustness in the design is achieved through the use of a hierarchical directory that grows in a manner similar to a multidimensional B-tree. For most practical directory sizes of at most 2^{32} entries, we guarantee no more than three disk accesses for an exact match search. Like the grid file, the directory corresponds to a rectilinearly partitioned attribute space which is represented as d-dimensional extendible array. Hence range and partial-range searches are efficiently executed in $O(n_R)$, where n_R is the number of rectangular cells that cover the response region.

1. Introduction

Given a file of records whose keys are composed of d-dimensional vectors $K = \langle k_1, k_2, \ldots, k_d \rangle$, we address the problem of the storage and maintenance of such a file under the operations of insertions, deletions and partial-range queries. Exact-match, partial-match and range queries are considered as special cases of the partial range queries. Let the attributes be defined by integers $1, 2, \ldots, d$ and let S be a subset of the attributes with cardinality $|S| \leq d$. For each attribute j, let $[\alpha_j, \beta_j]$ be some specified interval. Then the partial-range query recovers all records whose key $\langle k_1, k_2, \ldots k_d \rangle$ satisfy the predicate $\mathcal{F} = \bigwedge_{j \in S} (\alpha_j \leq k_j \leq \beta_j)$.

The interest in such multidimensional data organization is expressed in many applications of relational, geographic, pictorial and geometric databases that require extensive associative, and region searching. The literature is adorned with diverse strategies for implementing such data organization. They span from tree structured methods such as the K-D-tree [1], the Quadtrees [6,19,22] and the K-D-B-trees [21], to multidimensional direct access methods such as the grid-file or multidimensional extendible hashing [7,8,12,15,18,20,23], the interpolation-based index method [3], multidimensional linear hashing [17], and dynamic multipaging [14].

The direct access methods with the exception of dynamic multipaging, are all based on one or the other of two one dimensional dynamic hashing : extendible hashing [4] and linear hashing [9]. The two disk access principle of extendible hashing and the fact that no special overflow organization is required, make it very attractive for adaptation in multiple key data organization.

The underlying principle of the multidimensional extendible hashing is as follows. Let each key be a d-dimensional vector $K = \langle k_1, k_d, \ldots, k_d \rangle$. In the sequel, our use of the term "key" K may also imply a record with a key value K, depending on context. A pseudo-key K' corresponding to K is defined as $K' = \langle k'_1, k'_2, \ldots, k'_d \rangle$, where $k'_j = \psi_j(k_j)$, for $j = 1, 2, \ldots, d$, is a binary value in [0,1]. The function ψ_j defines a binary encoding of values of attribute j. Equivalently, k'_j may be conceived as an infinite sequence of 0/1 bits. Consider the geometric representation of the records of the file as points in a d-dimensional hypercube $[0,1]_1 \times [0,1]_2 \times \ldots \times [0,1]_d$. This space is termed the attribute space, bitmap or key space of the file. The dynamic direct access file organization schemes essentially partition the attribute space rectilinearly into rectangular regions or cells such that, for some predefined page capacity b, the number of points in a rectangular cell is no more than b. This explicit partitioned space is represented either as a directory whose entries

 ACM-0-89791-179-2/86/0300-0100 $00.75

are pointers to the pages where the records are stored or as data pages. In the former representation, the attribute space may be conceived as partitioned explicitly into regions of rectangular cells where each region corresponds to a group of cells with a common page pointer. The significant differences between the various dynamic multidimensional hashing schemes is in the implementation of the directory and the organization of the data pages.

To locate a record in two disk accesses, the directory is typically represented as a one level multidimensional array. It is shown in the analysis of extendible hashing and the grid file [5,11,20], that if the distributions of the keys in the attribute space is reasonably uniform, the directory size is superlinear. On the other hand non-uniform data distribution generates an almost exponential directory growth particularly for small data page capacities. The use of the scheme for efficient range searching requires order preservation in the sense that for each attribute j having key components $k_{j_1}$and k_{j_2}, if $k_{j_1} \leq k_{j_2}$ then $k'_{j_1} \leq k'_{j_2}$. Under such conditions one frequently encounters non-uniform data distributions and the directory growth becomes of much concern.

The question of organizing the directory for an order preserving extendible hashing scheme so that it gracefully adapts to the key distribution has only recently been addressed by Burkhard [3], Krishnamurthy and Whang [8], Ouksel [18] and Hinrichs [7]. There are two main ideas that have been advanced. The first advocates the use of a piecewise linear function. The second proposes the use of a two level directory. We present a general method for designing a multilevel directory associated with multidimensional extendible hashing that allow linear directory growth. We show that a straight forward hierarchical built up of the directory fails to restrain the potential exponential growth of the directory even for uniform key distribution.

In [18], we developed the multidimensional extendible hashing (MDEH), and defined appropriate mapping function for the scheme. We resolve the shortcomings in the design so that non-uniform distributed keys may be satisfactorily handled with a directory of moderate size. This new scheme is referred to as a Balanced Hierarchical Multidimensional Extendible Hash Tree (BMEH-tree). The method integrates the concepts of MDEH and the K-D-B-tree of Robinson [21]. Unlike any known methods, the local depth maintained in the directory play a significant role in determining the target page of a key. To highlight the considerable reduction in directory size under uniform and non-uniform key distributions we compare the scheme with the one level directory design (MDEH) and an alternative multilevel directory design whose nodes are not height balanced. This latter design is referred to simple as a multidimensional extendible hash tree (MEH-tree). Our experimental studies show that the BMEH-tree is clearly superior in maintaining the minimum directory size under both uniform and non-uniform key distribution.

2. Design Concepts

2.1. A Variant of Extendible Hashing

The progression of the ideas involved in the design of the balanced multidimensional extendible hashing scheme begin with an order preserving variant of the extendible hashing technique of Fagin et al. [4]. This is outlined briefly here. Let each key be a single attribute value and for simplicity, consider this as a sequence of 0/1 bits $K = x_1x_2x_3\ldots x_w$, of length w = 32 say. The file organization is comprised of two levels: a directory which we denote by D, and a set of data pages $\{P_1, P_2, \ldots, P_{n_p}\}$. The value n_p varies in consonance with the file expansion and contraction.

The directory is composed of a directory header $D.H$, called the file depth and n_d directory elements $D_1, D_2, \ldots, D_{n_d}$, where $n_d = 2^{D.H}$. Each directory element D_i, consists of a page pointer $D_i.P$ and a local depth $D_i.h$. The storage of the local depth as a component of the directory element differs from the original design in [4] where the local depth is maintained in the data pages. Storing the local depth in the direcotry allows an immediate deletion of empty pages. The local depth signifies the length of the common prefixes of the keys in a page. Suppose the binary bit sequence of a key $K = x_1x_2x_3\ldots x_w$. Then the address i of the directory entry D_i, corresponding to K, is determined by the first H prefix bits of K. This is defined by a function g as

$$i = g(K, H) = \sum_{1 \leq r \leq H} x_r 2^{H-r}.$$

The manner in which the file expands is illustrated using Figure 1a. Suppose the key K = "10101..." is to be inserted and the global depth $D.H = 2$. Using the prefix bits "10" we determine the address $g(K, D.H) = 2$. The page pointer in D_2 is given by $D_2.P = P_2$, and the local depth at this address is $D_2.h = 1$. Assuming the page P_2 is full then a third page P_3 will be allocated. The page pointers of the directory elements $D_2.P$ and $D_3.P$ are set to P_2 and P_3 respectively. Similarly the local

Figure 1a: 1-dimensional extendible hashing, H = 2

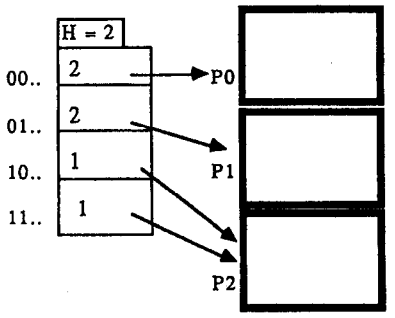
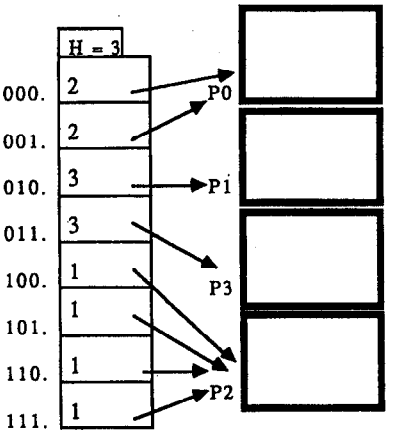


Figure 1b : 1-dimesional extendible hashing after directory expansion.



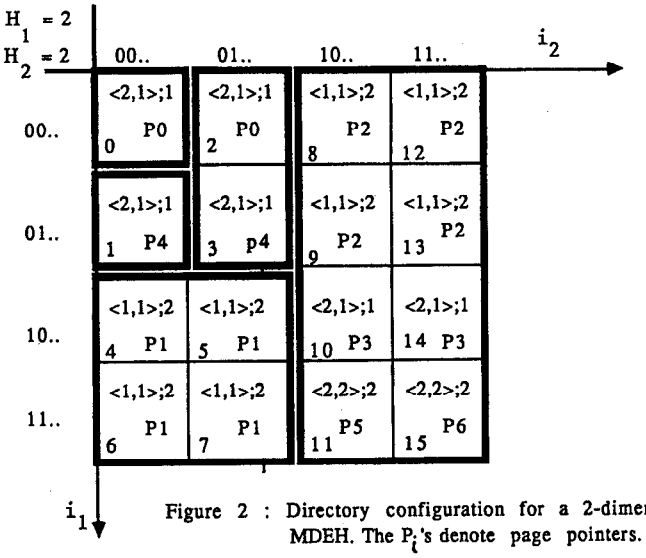


Figure 2 : Directory configuration for a 2-dimensional MDEH. The P_i's denote page pointers.

depths $D_2.h$ and $D_3.h$ are increased to 2. The keys in the page P_2 are rehashed to be distributed between P_2 and P_3.

If the key to be inserted were K = "01101...", then the directory element addressed would be D_1 and the page pointer $D_1.P = P_1$. Splitting the page P_1 would result in increasing the local depth $D_1.h$ to 3. Since this exceeds the global depth, the global depth $D.H$ is also increased to 3 with the consequence that the directory size is doubled. This gives the configuration shown in Figure 1b.

The extendible hashing scheme has been extensively analysed (see [5,11]). Under the assumption that, all possible configurations of allocating N keys into pages of capacity b are equally likely, the directory size in extendible hashing is shown to be

$O((1+1/b)\log_2 N)N^{1+1/b})$.

2.1. Multidimensional Extendible Hashing With One-Level Directory

Consider now that each key is a vector $K = \langle k_1, k_2 \ldots, k_d \rangle$, where $k_j = x_{1-j}x_{2_j}\ldots x_{w_j}$, is a binary sequence. As in the preceding section, the file is organized in two levels of a directory and set of data pages. The directory is headed by d global depths $H_1, H_2, \ldots, H_d$. A directory element D_i consist of d local depths $D_i.\langle h_1, h_2, \ldots, h_d \rangle$, a value $D_i.m$, specifying the dimension along which the last directory expansion was made, and a page pointer $D_i.P$. The relationship between the local and global depths are maintained independently for each dimension such that $D_i.h_j \leq D.H_j$, for $j = 1, 2, \ldots, d$.

To determine the page address of the key $K = \langle k_1, k_2, \ldots k_d \rangle$, we first compute d index values

$$i_j = g(k_j, H_j) = \sum_{1 \leq r \leq H_j} x_{r_j} * 2^{H_j - r_j}, \text{ for } j = 1, 2, \ldots, d.$$

These values form a d-tuple index $\langle i_1, i_2, \ldots i_d \rangle$ which is used to address the directory. Let $\mathcal{G}$ denote an appropriately defined mapping function. Then the address q, of a directory element D_q corresponding to the key K, is given by $q = \mathcal{G}(i_1, i_2, \ldots i_d)$. The required mapping function $\mathcal{G}$ is essentially that of an extendible array of exponential varying order which is described in detail in [15]. For completeness we restate it in Theorem 1.

Theorem 1. *The d-tuple index $\langle i_1, i_2, \ldots, i_d \rangle$ of a d-dimensional extendible array $A[0 : 2^{h_1}, 0 : 2^{h_2}, \ldots, 0 : 2^{h_d}]$, of exponential varying order in which each dimension has infinite extendibility can be mapped one-to-one onto the logical integer addresses $\{0, 1, 2, \ldots\}$, by the function $\mathcal{G}$ defined as*

$$\mathcal{G}(i_1, i_2, \ldots i_d) = \begin{cases} 0, & \text{if } \max_j(i_j) = 0; \\ i_z \prod_{\substack{j=1 \\ j \neq z}}^{d} J_j + \sum_{\substack{j=1 \\ j \neq z}}^{d} C_j * i_j, & \text{otherwise;} \end{cases}$$

where z = highest dimension index z s.t.;

$$\lfloor \log_2 i_z \rfloor = \max_j(\lfloor \log_2 i_j \rfloor),$$

$$J_j = \begin{cases} 2^{h_z+1} & \text{if } j < z; \\ 2^{h_z} & \text{if } j \geq z. \end{cases}$$

and $C_j = \prod_{\substack{r=j+1 \\ r \neq z}}^{d} J_r$

Considering the evaluation of the $\log's$ and exponentiations as primitive operations, the time complexity of $\mathcal{G}$ is $O(d)$. The function $\mathcal{G}$ is easily modified to take care of the case where a dimension does not have infinite extendibility and the cyclic choice of the dimensions skips over this. This corresponds to the case where the attribute values of a dimension may be coded by a shorter string of binary digits than the rest.

Given the directory element D_q, the page pointer is denoted by $D_q.P$. Assuming the page $D_q.P$ overflows. Then the keys in the page $D_q.P$ are split, with the consequence that the local depth $D_q.h_m$ of some dimension m, must be increased. Within the entry D_q, the value of m is updated before being used, and is updated cyclically by assigning $D_q.m \leftarrow (D_q.m \mod d) + 1$. If in increasing $D_q.h_m$, this value exceeds the global depth H_m, then the directory is doubled along the dimension m. The effect of the cyclic doubling of a 2-dimensional directory is illustrated by the Figure 2. The number indicated in each cell denotes the logical linear address of the cell.

3. Hierarchical Directories

The directory in the MDEH scheme above can easily degenerate giving a considerable large directory size for two main reasons. Most data distributions encountered in practice are highly non-uniform particularly when order preservation is enforced. Data that is being processed dynamically often exhibit some

noise effect, where a short burst of consecutive keys inserted differ only in the low order bits and therefore cause repeated splitting and directory expansion. Consider the one-dimensional extendible hashing scheme discussed in the preceding section. Assuming that each key is a 32-bit binary integer with maximum value $M = 2^{31} - 1$, then the worst case directory size after inserting $N \geq b + 1$ keys, is $O(M/(b+1))$. Further the cost of key insertions becomes $O(M/(b+1))$ directory accesses. This results from resetting half the number of page pointers in the directory when new page is allocated. It has been recommended that large page sizes be used since this tends to generate small directory sizes [11]. Another proposed solution to controlling the generation of large directory sizes is the use of a hierarchical directory. A design strategy for the case of single attribute keyed file is presented in [16]. The potential for the exponential growth of the directory is accentuated in the one-level multidimensional equivalent described previously. We present the balanced hierarchical multidimensional extendible hash tree (BMEH–tree) as a solution.

3.1. The Basic Idea of the BMEH–tree

The main idea of the scheme is to allocate the storage space of the directory in fixed size blocks or pages that form the nodes of a balanced M-ary tree. In general a block of the directory would be chosen to be of the same size as the data page. The leaf nodes in an ℓ-level BMEH–tree is taken as the first level nodes of the directory which is considered to be at level 1. The root node is at level ℓ. All data pages are at the same level, i.e level 0. Only the leaf nodes of the directory contain pointers to data pages. The directory elements of the higher level nodes contain pointers to lower level nodes. A number of pointers in a node can point to the same node at a lower level (see Figure 3c).

Let $\xi_1, \xi_2, \ldots, \xi_d$ be the maximum file depths of each dimension within a node, and let $\phi = \sum_{1 \leq j \leq d} \xi_j$. Then a node can contain at most $M = 2^{\phi}$ elements. Each node in the tree is organized as in the one-level directory of the multidimensional extendible hashing scheme except that the global depths H_j, of each node can only increase up to ξ_j. Any subsequent expansion of the node results in node splitting. For simplicity we illustrate the node splitting process for a 2-dimensional directory tree, where $d = 2$. Let $\xi_1 = \xi_2 = 2$. Starting with a single directory node D_1^1, the node doubles at each expansion step until $D_1^1.H_1 = \xi_1$ and $D_1^1.H_2 = \xi_2$.

Assuming that the key $K = \langle$"01011...", "10100..."$\rangle$, is to be inserted and the directory is to be doubled. The directory entry in D_1^1 determined by the key K, has the coordinate address $\langle$"01", "10"$\rangle = \langle 1,2 \rangle$. Let this be denoted as $D_1^1\langle 1,2 \rangle$. We illustrate this as the shaded cell in Figure 3a. Instead of doubling the node along the first dimensions so that the first three prefix bits are used in generating an index of the first dimensions, we split this node into two nodes, D_1^1 and D_2^1. A third node D_1^2, is created which now becomes the parents of of nodes D_1^1 and D_2^1. The two entries in the node D_1^2 are $D_1^2\langle 0,0 \rangle$ and $D_1^2\langle 1,0 \rangle$. The first entry points to the lower level node D_1^1 and the second points to D_2^1. Each directory entry within each node maintains appropriate information of the local depth and the dimension m, which has just been extended. The schematic storage layout is illustrated in Figure 3b.

In this scheme, the local depths play a significant role in determining the target page of any key K, as we traverse the directory tree. During the split, the local depth h_1 of every directory entry of the nodes D_1^1 and D_2^1 is decreased by one, except for two entries both of which are in either D_1^1 or D_2^1 that have pointers to the pages created from page splitting. If the page P_a in Figure 3a is the page that triggered the page splitting, then the two pages generated are P_a and P_b as shown in Figure 3b. At the same time the local depth h_1 in the two entries $D_1^2\langle 0,0 \rangle$ and $D_1^2\langle 1,0 \rangle$ are initialized to 1. The effect of such a splitting process is that the directory tree grows towards the root, in the manner reminiscent of the K–D–B–tree of Robinson [21]. Consequently the directory tree is completely balanced with respect to the path length from the root to a data page.

Carrying the example one step further in the expansion process, let us assume that the directory node D_1^1 is split as a result of a key insertion into page $D_1^1\langle 2,2 \rangle.P = P_a$ say. The resulting configuration of the scheme after the node splits is shown in Figure 3c. The node D_1^1 is split into two nodes D_1^1 and D_3^1 on the dimension 2. The creation of these two nodes causes the expansion of the parent node D_1^2 and the global depth $D_1^2.H_2$ increases to 1. The pointers in the directory entries of the root node are now defined as $D_1^2\langle 0,0 \rangle.P = D_1^1$, $D_1^2\langle 0,1 \rangle.P = D_3^1$ and $D_1^2\langle 1,0 \rangle.P = D_1^2\langle 1,1 \rangle.P = D_2^1$. At the first level of the directory tree, the two pages involved in the split are given by $D_1^1\langle 2,2 \rangle = P_a$ and $D_1^1\langle 2,3 \rangle.P = P_b$. The splitting process is carried out for each dimension in turn, cyclically, until the root node becomes full. This may generate further splitting and eventually cause the root node to split as well.

Figure 3a

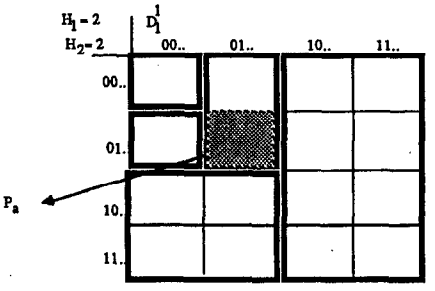


Figure 3b

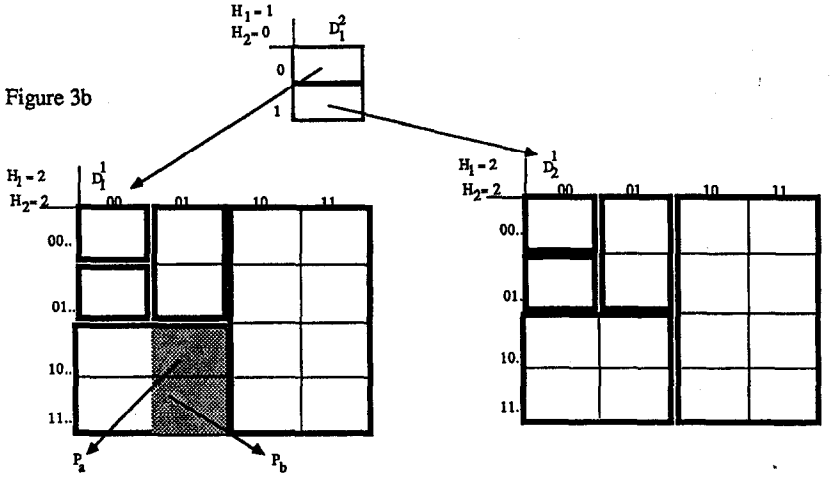
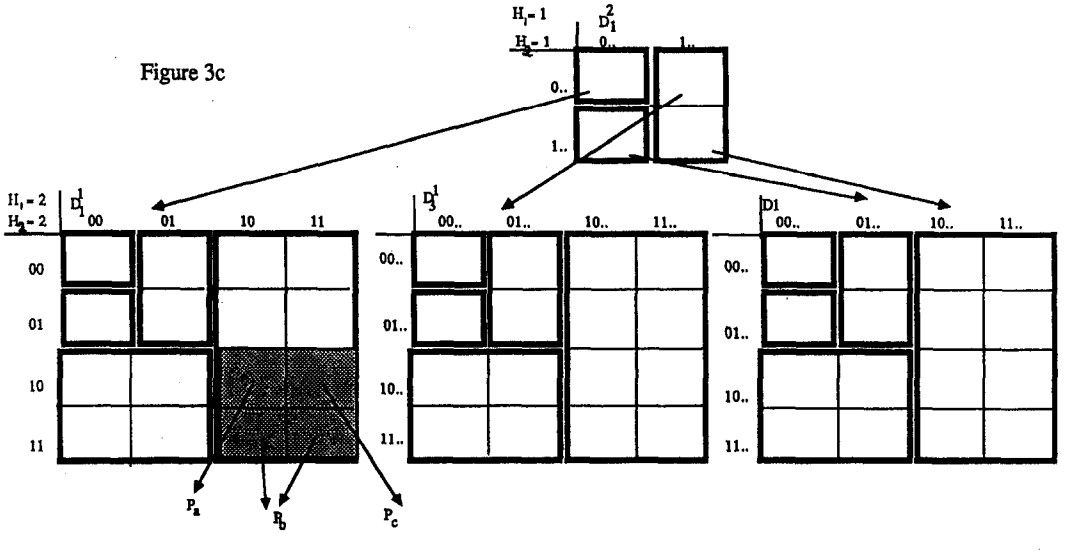


Figure 3c



To address the page of the key $K = (\text{"0101}\ldots\text{"}, \text{"1010}\ldots\text{"})$ in Figure 3c, the first bit "0", of the first component key value and bit "1" of the second component key determine the address of the entry $D_1^2\langle 0,1\rangle$ at the root node. The pointer in this entry is D_2^1. Suppose the local depths are $D_1^2\langle 0,1\rangle.h_1 = 1$ and $D_1^2\langle 0,1\rangle.h_2 = 0$. This implies that we strip off the first bit of the first key component but none from the second key component when determining the correct directory entry of node D_2^1. The directory entry from the rest of the key values, is generated as $D_2^1(\text{"10"}, \text{"10"})$ or $D_2^1\langle 2,2\rangle$. The page pointer of this directory element gives the target page of the key K as $D_2^1\langle 2,2\rangle.P = P_x$.

The algorithm for searching for a key $K = \langle k_1, k_2, \ldots k_d\rangle$ in a balanced hierarchical extendible hash tree is specified as follows. The variable ROOT specifies the address of the root node. We assume that the routine **GetPage**(P) retrieves into main memory, the page corresponding to either the directory node or data page pointer P and returns a pointer to the main memory address where the page resides.

Algorithm EXM_Search($\langle k_1, k_2, \ldots, k_d\rangle$, ROOT)
begin
  NodePtr ← GetPage(ROOT);
  $v_j \leftarrow k_j$ for $j = 1, 2, \ldots, d$;
  Set $i_j \leftarrow g(v_j, NodePtr \uparrow .H_j)$, for $j = 1, 2, \ldots d$;
  Set $q \leftarrow \mathcal{G}(i_1, i_2, \ldots, i_d)$;
  Set $P \leftarrow NodePtr \uparrow .D\langle q\rangle.P$;
  if P is a directory node pointer **then**
  begin
    Remove the first $NodePtr \uparrow .D\langle q\rangle.h_j$ bits from
    v_j, for $j = 1, 2, \ldots, d$;
    EXM_Search($\langle v_1, v_2, \ldots, v_d\rangle, P$);
  end else
  begin
    Read the page P into memory and search for
    the key K;
    if found **then** return ("found")
    else return("not found");
  end;
end;

Let $\phi = \sum_{1 \le j \le d} \xi_j$ denote the sum of the global depths allowed in a node. If the total number of prefix bits used in addressing a directory of size n_d, is w, then the maximum number of levels in the tree $\ell = \lceil w/\phi \rceil$. For instance, choosing $\phi = 9$ gives $\ell \le 3$ for $w \le 27$ and $\ell \le 4$ when $w \le 36$. Considering that the root node can always be retained in memory, it takes at most 3 disk accesses to locate a record of a file with a directory size $n_d \le 2^{27}$.

4. Insertions, Deletions and Retrievals

4.1. Insertions

The discussions in the preceding section leads to the following algorithm "BMEH-Tree_Insert" for inserting a record with key $K = \langle k_1, k_2, \ldots, k_d\rangle$ into a balanced multidimensional extendible hashing structure. The algorithm has some dependent routines whose functions only are described. We assume that a global stack "STACK" is available.

Push (NodePtr, I, STACK) : This routine pushes the node pointer NodePtr and the index I into a stack STACK.

Pop (NodePtr, I, STACK) : The function of this routine is to pop the values I and NodePtr from the stack STACK.

Expand_Dir(Node, P_1, P_2, m) : The routine expands the directory node in a manner corresponding to that of an extendible array of exponential varying order. The pointer fields of two entries in this node are set to contain P_1 and and P_2.

Split_Node (Node, P_1, P_2, m) : This routine splits the node given by Node. The pointer fields of two entries in one of the new nodes are updated with the values of P_1 and P_2. The values P_1 and P_2 are respectively reset to Node and the new allocated page involved in the split. The routine returns false if the ROOT node is split.

Algorithm **BMEH_Insert**(K = $\langle k_1, k_2, \ldots, k_d\rangle, ROOT$);
begin
  $v_j \leftarrow k_j$, for $j = 1, 2, \ldots, d$;
  P ← ROOT;
  Initialize the STACK to empty;
  while P is a directory pointer **do**
    NodePtr ← GetPage(P);
    $i_j \leftarrow g(v_j, NodePtr \uparrow .H_j)$;
    $q \leftarrow \mathcal{G}(i_1.i_2, \ldots, i_d)$;
    Push(P, q, STACK);
    Left_Shift $(v_j, NodePtr \uparrow .D\langle q\rangle.h_j); j = 1, 2 \ldots d$;
    $P \leftarrow NodePtr \uparrow .D\langle q\rangle.P$;
  endwhile;
  if P = NIL **then**
  begin
    allocate a new data page P_K;
    Using the value of $NodePtr \uparrow .D\langle q\rangle.m$ and
    the difference between the global depth $NodePtr \uparrow .H_m$
    and the local depth $NodePtr \uparrow .D\langle q\rangle.h_m$,
    determine all directory entries having the same
    file depths as the one given in $NodePtr \uparrow .D\langle q\rangle$,
    and set the pointer field values to P_K.
    Store the key in page P_K and return;
  end else
  begin
    $P_1 \leftarrow GetPage(P)$; { read into memory page P }
    if page P already contains key K **then**
      print ("error message") and return;
    if page P is not full **then**
      store record in page P

```
else begin
  allocate a new page P_K;
  set P_2 ← GetPage(P_K);
  copy content of page P into a temporal
  storage Q;
  Set Cont_Split ← true;
  while Cont_Split do
    Pop (NodePtr, q, STACK);
    if number of entries in NodePtr ↑ .D ≤ 2^φ
    then begin
      Expand_Dir(NodePtr ↑, P_1, P_2, m);
      Set Cont_Split ← false;
    end else
      Cont_Split ← Split_Node (NodePtr ↑,
      P_1, P_2, m);
  endwhile
  Batch insert all keys in Q;
  BMEH_Insert(K = ⟨k_1, k_2, ..., k_d⟩, ROOT);
 endif
endif
 end { algorithm BMEH_Insert };
```

The BMEH–tree organization allows the worst case number of directory splitting and worst case number of directory accesses per record insertion to be controlled according to the choice of the values $\xi_1, \xi_2, \ldots \xi_d$. We have the following theorems which we state below.

Theorem 2. *In a balanced d dimensional extendible hash tree with parameters d, b, w, ξ_j, for $j = 1, 2, \ldots d$, let $\phi = \sum_{1 \le j \le d} \xi_j$ be the specified bound on the number of bits allowed for addressing within a node of the BMEH–tree. Then for a directory using at most w bits to address an entry, the worst case number of node splits for an insertion is $\frac{\ell(\ell-1)}{2}\phi + l$, where $\ell = \lceil w/\phi \rceil$.*

Sketch of Proof

Let the maximum number of bits used in addressing a page in a the BMEH-Tree organization be w and define $\phi = \sum_{j=1}^{d} \xi_j$. The maximum number of levels in the directory is $\ell = \lceil w/\phi \rceil$. For simplicity we assume that all the key components participate in the address calculation with equal number of bit encodings. The worst case number of splits occurs when, in attempting to insert the (b+1)st key, all the keys agree in the first w-1 bits compared but has at least one key that differs on the last bit.

The directory tree generated is such that there is one root node on level ℓ, $(\phi + 1)$ nodes on level $\ell - 1$, $(2\phi + 1)$ nodes on level $\ell - 2$ and so on. On level i, there are $(\ell - i)\phi + 1$ nodes. The total number of nodes in the directory then is

$$n_d = 1 + (\phi + 1) + (2\phi + 1) + \cdots + (\ell - 1)\phi + 1,$$
$$= \frac{2 + (\ell - 1)\phi}{2}\ell = \frac{\ell(\ell - 1)\phi}{2} + \ell.$$

Since these n_d nodes are generated from $n_d - 1$ splits plus one extra from the page split, the worst case number of node splits is given by $(\frac{\ell(\ell-1)\phi}{2} + \ell)$. ∎

Theorem 3. *In a balanced d dimensional extendible hash tree with parameters d, b, w, ξ_j, for $j = 1, 2, \ldots d$, let $\phi = \sum_{1 \le j \le d} \xi_j$ be the specified bound on the number of bits allowed for addressing within a node of the BMEH–tree. Then for a directory using at most w bits to address an entry, the worst case number of directory node accesses for an insertion is $O(\phi\ell^2)$, where $\ell = \lceil w/\phi \rceil$.*

The argument for the proof of Theorem 3 follows from Theorem 2.

4.2. Deletion

The splitting process is easily reversed to handle deletions. The nodes may be recursively merged, starting from the bottom until possibly the root node is deleted. The deletions process does no encounter the problem of deadlocks as in the grid–file [7, 12], since we adhere strictly to the reversal of the insertion process. The details of how this is performed is easily derived from the insertion algorithm.

4.3. An Example

The following examples illustrates the essential concepts of the BMEH–tree. Consider the storage of the 2-dimensional keys of Table 1 using balanced hierarchical 2-dimensional hash tree.

Suppose the parameters specified are $\xi_1 = \xi_2 = 2$, and the page capacity $b = 2$. Then the Figure 4 shows the configuration of the scheme after all key insertions are made. The explicit attribute space partitioning induced in this case is shown in Figure 5.

An alternative to the BMEH–tree is another tree structured directory which we refer to as a multidimensional extendible hash tree (MEH-tree). In this scheme the tree grows from the root downwards. Although this design is simpler to implement, the reduction in the directory size is not significant compared to the one-level directory design of a multidimensional extendible hashing scheme (MDEH). In some instances and even for uniform distributed keys, the directory size in an MEH-tree structure can be worse than the one-level directory. This scheme has been implemented for comparison with the BMEH–tree.

Table 1 : A set of binary encoded keys.

K1	= (1110, 010)	K12	= (0111, 001)
K2	= (1011, 101)	K13	= (0011, 000)
K3	= (0101, 101)	K14	= (1100, 000)
K4	= (1100, 101)	K15	= (1001, 011)
K5	= (0001, 111)	K16	= (1101, 001)
K6	= (0010, 100)	K17	= (0011, 100)
K7	= (0100, 010)	K18	= (1110, 011)
K8	= (0111, 100)	K19	= (0111, 011)
K9	= (0001, 001)	K20	= (0001, 010)
K10	= (0110, 010)	K21	= (1001, 001)
K11	= (1000, 110)	K22	= (0110, 011)

4.4. Partial Range Retrievals

The BMEH-tree facilitates the processing of partial–range queries. Let S be a subset of the integers $\{1, 2, \ldots, d\}$, representing the dimensions in the scheme. For each $j \in S$, let $[\alpha_j, \beta_j]$ be a specified interval. Suppose we desire the set of records whose keys $K = \langle k_1, k_2, \ldots, k_d \rangle$ satisfy the predicate $\mathcal{F} = \bigwedge_{j \in S} (\alpha_j \leq k_j \leq \beta_j)$. Then the algorithm PRG_Search recursively traverses the directory node in depth–first–search order to retrieve the records in the pages whose corresponding cells are covered by the query region. We assume the existence of an order preserving binary encoding function ψ, and a procedure Left_Shift(x,y) which shifts the bits in x, y places to the left. The algorithm takes a parameter *ROOT*, which is the address of the root node of the directory tree, and pairs of binary integers k_{j_l}, k_{j_u} (one pair for each dimension), which are defined as follows.

$$k_{j_l} = \begin{cases} \psi(\alpha_j), & \text{if } j \in S; \\ \text{“00000...”}, & \text{otherwise.} \end{cases}$$

$$k_{j_u} = \begin{cases} \psi(\beta_j), & \text{if } j \in S; \\ \text{“11111...”}, & \text{otherwise.} \end{cases}$$

Algorithm PRG_Search$(\langle k_{1_l} : k_{1_u}, \ldots, k_{d_l} : k_{d_u} \rangle, \text{ROOT})$;
begin
  NodePtr ← GetPage(ROOT);
  Set $v_j \leftarrow k_{j_l}; u_j \leftarrow k_{j_u}; i_j \leftarrow L_j \leftarrow g(v_j, NodePtr \uparrow .H_j)$, and $U_j \leftarrow g(k_{j_u}, NodePtr \uparrow .H_j)$, for $j = 1, 2, \ldots, d$;
  Set Search_Region ← “true ”;
  while Search_Region **do**
  begin
    $q \leftarrow \mathcal{G}(i_1, i_2, \ldots, i_d)$; $P \leftarrow NodePtr \uparrow .D\langle q \rangle .P$;
    if P has not been accessed **then**
    begin
      if P is not a data page **then**
      begin
        Left_Shift$(v_j, NodePtr \uparrow .D\langle q \rangle .h_j)$, and
        Left_Shift$(u_j, NodePtr \uparrow .D\langle q \rangle .h_j)$,
        for $j = 1, 2 \ldots d$;
        PRG_Search$(\langle v_{1_l} : u_{1_u}, \ldots, v_{d_l} : u_{d_u} \rangle, P)$;
      end else
        Retrieve all records that satisfy the range predicate $\mathcal{F}$;
    end ;
    Set $j \leftarrow 0$, and Search_Region ← “false”;
    while $j \leq d$ and NOT Search_Region **do**
    begin
      $i_j \leftarrow i_j + 1$;
      if $i_j > U_j$ **then**
        begin Set $i_j \leftarrow L_j$; $j \leftarrow j + 1$; **end**
      else
        Search_Region ← “true”;
    end;
  end;
end;

Theorem 4 In a balanced d dimensional extendible hash tree with parameters d, w, b, ξ_j for $j = 1, 2, \ldots d$, let $S \subset \{1, 2, \ldots, d\}$. *Then a a query that requests the retrieval of all records with key $K = \langle k_1, k_2, \ldots k_d$ satisfying the predicate $\mathcal{F} = \bigwedge_{j \in S} (\alpha_j \leq k_j \beta_j)$ can be processed in $O(\ell n_R)$ disk accesses, where n_R is the number of rectangular cells of the partitioned space that cover the query region and $\ell = \lceil w/\phi \rceil$.*

Sketch of Proof

Consider the rectilinear partitioning of the attribute space induced by the BMEH-tree. For an orthogonal range query, the query region is overlapped by a number of such cells. In the worst case each cell contains a pointer to a separate data page which can be arranged to be accessed once. If the total number of cells covering the query region is n_R and the cost of accessing each page is at most ℓ then we require $O(\ell * n_R)$ to retrieve the requested records. ∎

5. Experimental Results

As a first step towards understanding the behaviour of the balanced multidimensional extendible hash tree organization, we study some performance characteristics through simulation.

These measures are compared with those of the multidimensional extendible hashing with one–level directory (MDEH), and the multidimensional extendible hash tree (MEH-tree) for data page sizes of 8, 16, 32 and 64.

The experiments are conducted for two classes of data distributions :

1. uniform distributed keys in which each key component is a pseudo random integer in $[0, 2^{31} - 1]$ (we investigate this for 2- and 3-dimensional keys);
2. a two dimensional (bivariate) normal distributed keys where each component of the key vector is a truncated discretized normal in $[0, 2^{31} - 1]$.

Each run of the experiment consists of inserting N = 40,000, keys and computing the averages of the performance measures on the last 4,000 keys inserted. In the BMEH–tree and the MEH-tree, the node sizes are restricted to 64 entries only, i.e., $\phi = 6$. For $d = 2$, we have $\xi_1 = \xi_2 = 3$, and for $d = 3$, we have $\xi_1 = \xi_2 = \xi_3 = 2$. This is to allow for a fast build up of the number of directory levels. The performance parameters derived are :-

- λ : the average number of disk reads for a successful exact–match search.
- λ' : the average number of disk reads for an unsuccessful exact–match search.
- ρ : the average number of disk accesses for a key insertion. We consider a disk access as either a disk read or a write.
- σ : the directory size (in number of directory elements) generated after 40,000 key insertions.
- α : the average load factor which is defined as the ratio of the number of keys inserted to the amount of storage space made available by data pages allocated.

The result of the simulations are summarized in the Tables 2, 3, and 4. In Figures 6 and 7, we show the graphs of the variation of the directory size (n_d) as random keys are inserted for the two cases of 2-dimensional uniform and non-uniform distributed keys. The BMEH-tree is clearly superior in maintaining a much smaller directory size in either case. Further the directory grows almost linearly with the number of keys inserted.

6. Conclusion

Using the balanced multidimensional extendible hash tree technique gives us a new method of data organization that improves upon the one-level directory method of multidimensional extendible hashing and the grid-file. The method inherently controls the possible exponential growth of the directory without compromising on the $O(1)$ disk access principle guaranteed in extendible hashing schemes. Not only does the BMEH-tree maintain an almost linear growth for both uniform and non–unform data distribution, the average number of disk accesses for a key insertion is considerable less than in the MDEH scheme. We draw the readers attention particularly to the value of ρ in Table 3 when $b = 8$.

The ideas in the BMEH–tree may be extended to generate another breed of tree structures that may be characterized as Balanced Binary Quadtree, Octtree etc. This is easily achieved by setting $\xi_j = 1$, for every dimension and deleting some of the information retained in the directory elements. The standard Quadtree [19, 22] and its derivatives have previously been known to be difficult to balance. The BMEH-tree is a natural candidate for the physical design of such data base systems as in relational, geographic , geometric, pictorial and CAD databases, whose applications require a high degree of associative or spatial searching.

Acknowledgment

We wish to express our thanks to the department of Computer and System Engineering at Carleton University for the use of their VAX/780 in running our simulations. The independent implementation of this work by George Wang for solving some Geometric problems is very much appreciated. This research is supported in part by the Natural Sciences and Engineering Research Council of Canada under grant No A0317–102B.

References

[1] Bentley,J. L. Multidimensional binary search tree in database organization. *IEEE Trans. on Soft. Eng., SE-5, 4 (1979), 333–340.*

[2] Bayer, R. and McCreight, E. Organization and maintenance of large ordered indexes. *Acta Informatica, 1, 3 (1972), 173–189*

[3] Burkhard, W. A. Index maintenance for non–uniform record distribution. *Proc. SIGACT–SIGMOD Symp. on Principles of Database Syst., Waterloo, Canada 1984), 173–180.*

[4] Fagin, R., Nievergelt, J., Pippenger, N. and Strong, H. R. Extendible hashing: a fast access method for dynamic files. *ACM Trans. on Database Syst. 4, 3, (1979), 315–344.*

[5] Flajolet, P. On the performance evaluation of extendible hashing and trie searching. *Acta Informatica 20 (1983), 345–369.*

[6] Finkel, R. A. and Bentley, J. L. Quad trees : a data structure for retrieval on composite keys. *Acta Informatica, 4 (1974), 1–9.*

[7] Hinrichs, K., The grid file system : implementation and case studies of applications. *Ph. D Dissertation, Swiss Federal Institute of Technology, Zurich (1985).*

[8] Krishnamurthy, R. and Whang, K. Multilevel grid file. *Draft Report, IBM Research Lab., Yorktown Heights*

[9] Litwin, W. Linear hashing: a new tool for table and file addressing. *Proc. 6th Int'l. Conf. on Very Large Databases, Montreal (1980), 212–223.*

[10] Lomet, D. B. A high performance universal key associative access method. *Proc. ACM SIGMOD Conf., San Jose, (1983). 120–132*

[11] Mendelson, H. Analysis of extendible hashing. *IEEE Trans. on Soft. Eng. SE-8, 6 (1982), 611–619.*

[12] Nievergelt, J., Hinterberger, J. and Sevcik, K. C. The grid file : an adaptive symmetric multikey file structure. *ACM Trans. on Database Syst., 9, 1 (1984), 38–71.*

[13] Orenstein, J. and Merrett, T. H. A class of data structure for associative searching. *Proc. of ACM SIGACT-SIGMOD Symp. on Principles of Database Syst., Waterloo, Canada (1984), 181–190.*

[14] Otoo, E. J. and Merrett, T. H. Dynamic multipaging : a storage structure for fast associative searching. *Tech. Report No SCS54, School of Computer Science, Carleton University, Ottawa.*

[15] Otoo, E. J. A mapping function for the directory of a multidimensional extendible hashing. *Proc. 10th Int'l Conf. on Very Large Databases, Singapore (1984), 493–506.*

[16] Otoo, E. J. Linearizing the directory growth in extendible hashing. *Techinical Report No SCS77, School of Computer Science, Carleton University, August, 1985.*

[17] Ouksel, M. and Scheuermann, P. Storage mapping for multidimensional linear dynamic hashing. *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Syst., Atlanta, Georgia (1983), 90–105..*

[18] Ouksel, M. The interpolation-based grid file. *Proc. of fourth ACM SIGACT-SIGMOD Symp. on Principles of Database Syst., Portland, Oregon (1985), 20–27.*

[19] Overmars, M. H. and Leuween, J. Dynamic multidimensional data structures based on quad- and K-D- trees. *Acta Informatica, 17 (1982), 267–285.*

[20] Regnier, M. Analysis of grid file algorithms. *BIT, 25 (1985), 335–357.*

[21] Robinson, J. T. The K-D-B-tree : a search structure for large multidimensional dynamic indexes. *Proc. ACM SIGMOD Conf. Ann Abor, Michigan (1981), 10–18).*

[22] Samet, H. The quad tree and related hierarchical data structures. *ACM Comput. Survey, 16, 2 (1984), 187–260.*

[23] Tamminen, M. The extendible cell method for closest point problem. *BIT 22 (1982), 24–41.*

Figure 4 : The BMEH-tree obtained after inserting the keys of Table 1.

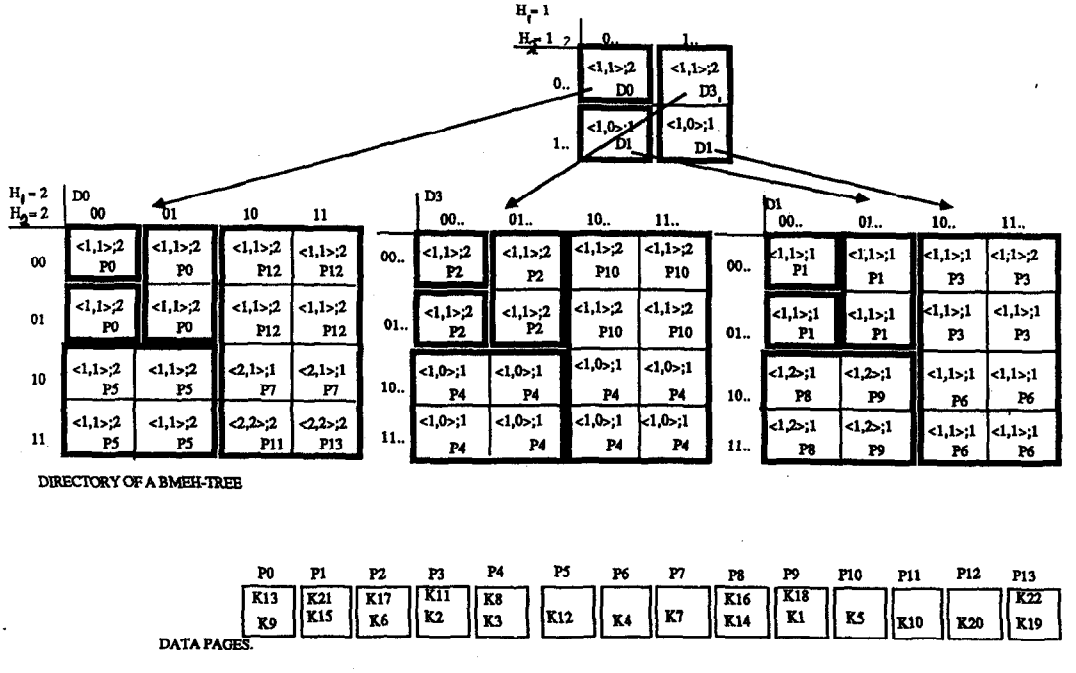


Figure 5 : The partitioned attribute space induced by the BMEH-tree.

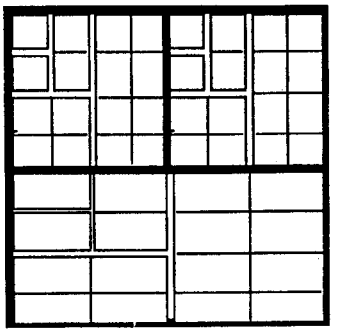


Table 2: Results for 2-dimensional uniform distributed keys.

Performance measure	Method of Ext. Hashing	Page Capcity , b			
		8	16	32	64
Avg. Disk I/O per succ. search, λ	MDEH	2.000	2.000	2.000	2.000
	MEH-Tree	2.756	2.039	2.000	2.000
	BMEH-Tree	3.000	3.000	2.000	2.000
Avg. Disk I/O per unsucc. search. λ'	MDEH	2.000	2.000	2.000	2.000
	MEH-Tree	2.574	2.011	2.000	2.000
	BMDEH-Tree	3.000	3.000	2.000	2.000
Avg. Disk I/O per. insertion ρ	MDEH	11.847	6.292	5.571	4.955
	MEH-Tree	6.198	4.110	3.503	3.256
	BMDEH-Tree	7.213	5.646	3.715	3.346
Avg. load factor, α	MDEH	0.692	0.682	0.658	0.626
	MEH-Tree	0.692	0.682	0.658	0.626
	BMEH-Tree	0.692	0.682	0.658	0.626
Dirctory Size for 40,000 insertions, σ	MDEH-Tree	65,536	8,192	4,096	1,024
	MEH-Tree	171,264	10,432	4,160	4,160
	BMEH-Tree	17,984	7,296	2,560	1,088

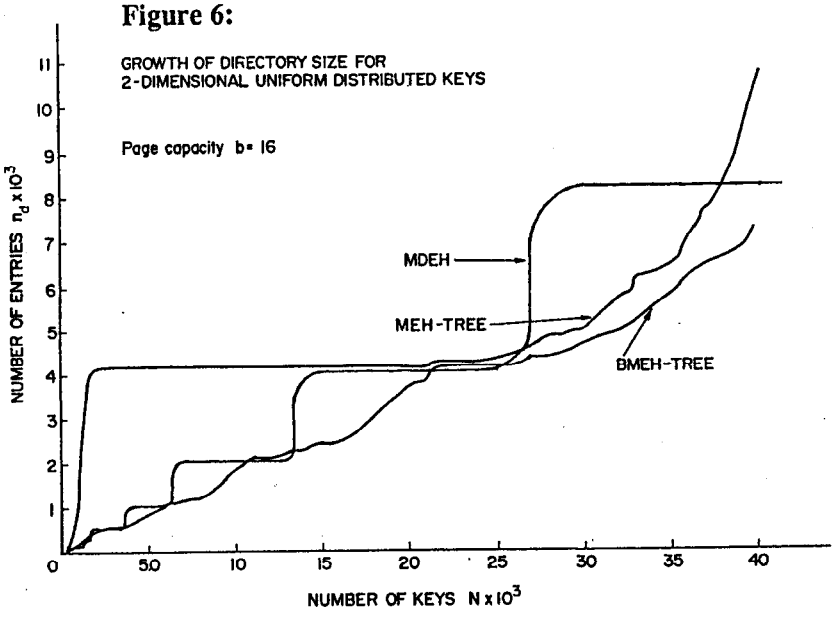
Table 3: Results for 2-dimensional normal distributed keys.

Performance measure	Method of Ext. Hashing	Page Capcity , b			
		8	16	32	64
Avg. Disk I/O per succ. search, λ	MDEH	2.000	2.000	2.000	2.000
	MEH-Tree	2.924	2.844	2.670	2.342
	BMEH-Tree	4.000	3.000	3.000	3.000
Avg. Disk I/O per unsucc. search. λ'	MDEH	2.000	2.000	2.000	2.000
	MEH-Tree	2.908	2.824	2.642	2.303
	BMEH-Tree	3.836	3.000	3.000	3.000
Avg. Disk I/O per. insertion ρ	MDEH	229.34	11.252	11.275	11.359
	MEH-Tree	6.267	4.971	4.241	3.615
	BMEH-Tree	8.415	5.523	4.804	4.427
Avg. load factor, α	MDEH	0.692	0.684	0.682	0.669
	MEH-Tree	0.692	0.684	0.682	0.669
	BMEH-Tree	0.692	0.684	0.682	0.669
Dirctory Size for 40,000 insertions, σ	MDEH	524,288	65,536	32,768	16,384
	MEH-Tree	66,368	48,896	30,848	13,440
	BMEH-Tree	20,800	9,856	5,248	2,624

Table 4: Results for 3-dimensional uniform distributed keys.

Performance measure	Method of Ext. Hashing	Page Capcity , b			
		8	16	32	64
Avg. Disk I/O per succ. search, λ	MDEH	2.000	2.000	2.000	2.000
	MEH-Tree	2.760	2.052	2.000	2.000
	BMEH-Tree	3.000	3.000	2.000	2.000
Avg. Disk I/O per unsucc. search. λ'	MDEH	2.000	2.000	2.000	2.000
	MEH-Tree	2.586	2.019	2.000	2.000
	BMEH-Tree	3.000	3.000	2.000	2.000
Avg. Disk I/O per. insertion ρ	MDEH	9.394	7.264	5.738	4.995
	MEH-Tree	6.184	4.129	3.567	3.253
	BMEH-Tree	7.343	5.771	3.757	3.353
Avg. load factor, α	MDEH	0.689	0.680	0.655	0.621
	MEH-Tree	0.689	0.680	0.655	0.621
	BMEH-Tree	0.689	0.680	0.655	0.621
Dirctory Size for 40,000 insertions, σ	MDEH	32,768	16,384	4,096	1,024
	MEH-Tree	170,752	10,688	4,160	4,160
	BMEH-Tree	17,984	8,000	2,432	1,088

Figure 6:



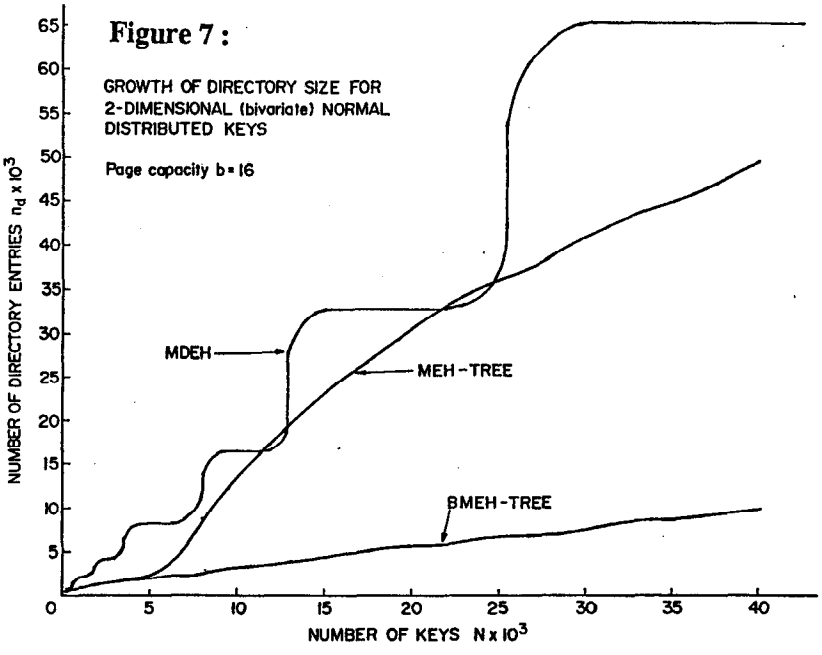


Figure 7 :
GROWTH OF DIRECTORY SIZE FOR
2-DIMENSIONAL (bivariate) NORMAL
DISTRIBUTED KEYS
Page capacity b = 16
NUMBER OF DIRECTORY ENTRIES $n_d \times 10^3$
NUMBER OF KEYS $N \times 10^3$
MDEH
MEH-TREE
BMEH-TREE
0
5
10
15
20
25
30
35
40
45
50
55
60
65