A MAPPING FUNCTION FOR THE DIRECTORY OF A MULTIDIMENSIONAL EXTENDIBLE HASHING

Ekow J. Otoo

March 1984

School of Computer Science
Carleton University
Ottawa, Ontario
Canada K1S 5B6

This research was supported by the Natural Sciences and Engineering Research Council of Canada.

A Mapping Function for the Directory of a Multidimensional Extendible Hashing.

Ekow J. Otoo

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

Abstract

A generalization of the Extendible Hashing scheme of Fagin and others is presented for structuring files of records with d-attribute fields. This generalization reduces to the problem of defining a storage mapping for an extendible array with exponential varying order. We define such a function with element address computation in time O(d), and we show how the result applies to the design of a multidimensional extendible hashing. Algorithms for searching, inserting and processing partial-match queries are presented and we discuss some peculiar characteristics of the scheme derived primarily by simulation studies done with both uniform and nonuniform distributed data.

1. Introduction.

Retrieval of records in current database systems often is by specifying one or more combination of the attribute values of the records being sought. Such retrieval operations have been termed associative searching or partial-match retrieval. Furthermore, some operational requirement usually dictate that the records be retrieved ordered on some fields. Given that the file is structured in some "ideal" form for efficient retrieval, we may require that the records resulting from operations on the original file be directly structured using the "ideal" form. This implies that a required characteristic of an "ideal" file organization is that it be dynamic; i.e., the file organization should tolerate insertions and deletions without compromising its retrieval efficiency. This requirement is critical in a highly volatile database environment.

Combined with the fact that hashing techniques provide the fastest direct access means to data on secondary storage, we postulate that a "multidimensional order preserving dynamic hashing scheme" satisfies the retrieval requirements of an "ideal" file organization if one can be realized. Although the above criteria are not satisfied by any one file organization method, a number of recent advances, satisfy more than one of the above requirements.

The various methods developed for efficient associative searching and partial-match retrieval may be categorized into two broad classes : a) direct access method as in the file design schemes in [1, 4, 5, 14, 16, 19, 20, 23, 24, 27, 30, 32], and b) tree structured file design methods as in [3, 10, 28, 30]. These two classes, however have a common geometric perspective of the file.

Consider a file $F = (r_1, r_2, \ldots, r_N)$, of N records, each record $r_i = (k_0, k_1, \ldots, k_{d-1})$, being defined by d attribute values where record retrievals are made by specifying some s combination of attribute values for $0 \leq s \leq d - 1$. A geometric interpretation of the file F is as a set of points in a d-dimensional space where each attribute corresponds to an axis of the space and the natural ordered distinct attribute values define descrete points along the respective axes. The presence of a record $r_i = (k_0, k_1, \ldots, k_{d-1})$ is denoted by a point at the coordinate point defined by the values $k_0, k_1, \ldots, k_{d-1}$. Such a representation is referred to as a multiattribute space model of the file. The Figure 1.1 illustrates the multiattribute space model of the file shown in its tabular form in Table 1.1.

Given such a file representation, one easily identifies that the proposed storage scheme for efficient partial-match searching effectively partition the multiattribute space model of the file into rectangular cells. Differences exist in the manner of splitting the space, and in the techniques for storing and retrieving the records whose point images fall in the cells. For example, methods such as multiple key hashing [4, 30], and multipaging [19, 22], assume a rectilinear partitioning of the space as illustrated by Figure 1.2. Another alternative rectangular, but non-rectilinear, partitioning is shown in Figure 1.3. Chang and colleagues [7], refer to such file organizations as having the cartesian product property. We may further distinguish the direct access methods according to whether the cells represent data pages as in [4, 19, 22, 23, 24], or whether they represent directory elements of page pointers as in [14, 16, 20, 30, 32].

In the tree structured design schemes, the partitioning boundary values of the space serve as discreminating values for the internal nodes of an index tree whose terminal nodes are pointers to the pages holding the records. Such design methods are exemplified by the heteregeneous K-D-Tree [3] and Quad-Tree [10]. Tree structured indexes inherently have the dynamic property except for the fact that they can degenerate with insertions and deletions. Since B-Trees [2], are most versatile in maintaining the balance of trees indexes in a dynamic environment the B-Tree design technique has been applied in multidimensional tree structures as the K-D-B-Tree of Robinson [28] and the multidimensional B-Tree of Sheuermann and Ouksel [31]. In a file of n pages, a record retrieval costs O(log n) page accesses using such tree structured organization. In this respect multidimensional direct access schemes, with O(1) page accesses to retrieve a record, appear more attractive. We focus attention then on multidimensional hashing schemes with dynamic characteristics. A number of different approaches to the design of a dynamic hashing schemes for 1-dimensional keys have been advanced by Larson [12, 13], Litwin [15], Fagin and others [9] and Lomet [17,18].

The various methods of dynamic multiattribute hashing schemes are adaptations of either Linear Hashing [12, 15] or Extendible Hashing [9]. For instance the Interpolation Based Index of Burkhard [5], the Extendible Cell method of Tamminen [31] and the Partial-Match Retreival technique of Lloyd and Ramamohanarao [16] all adapt extendible hashing by generating a single key from bit shuffling of the binary encoding of the multiple key values.

Where the attribute values of the records are independently considered, the design problem of a multidimensional dynamic hashing scheme reduces to defining a mapping function for extendible arrays with prescribed order of expansion. This is evident in the multidimensional linear hashing method of Ouksel and Scheuermann [24], the Grid File Method of Nievergelt and colleagues [20] and the multidimensional hashing schemes presented in [22,23].

The technique being presented in this paper is similar to the Grid File method in the sense that each attribute value is independently treated, i.e., the individual binary values of the attributes are not shuffled to form a single key. The main distinction of the method from earlier proposed schemes are :-

a) The directory is implemented as a d-dimensional extendible array with exponential expansion.

b) We define a mapping function for addressing the directory entries.

c) Empty pages resulting from page splits are eliminated.

In our discussions we will use the term "key" to refer to an attribute or a combination of attributes of the file whose values uniquely identify each record, and we shall refer to the content of a file as keys although each record may be comprised of a set of key fields and either the rest of the fields or a pointer to a loaction where the complete record is stored. In the rest of the paper, we present a review of extendible hashing in section 2 and an overview of a design of a

F (A_0,	A_1)
c	5
b	7
e	0
p	0
c	1
f	2
h	4
k	6
b	1
k	2
h	7
n	5
e	6
b	4
p	4

Table 1.1 : A table form of a file on 2 attributes A_0, A_1.

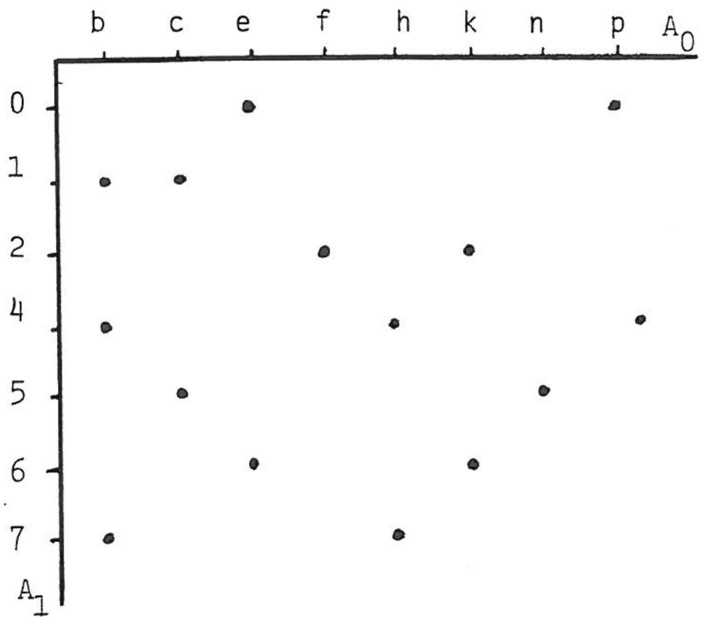

Figure 1.1 : A multiattribute space model of the file shown in Table 1.1.

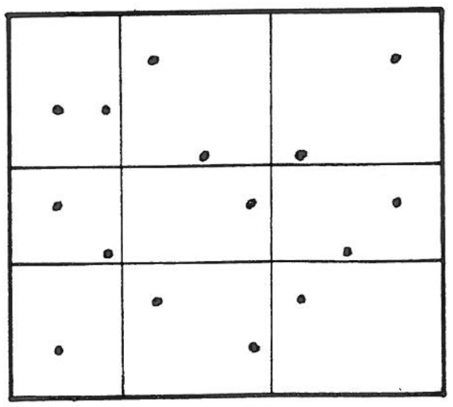

Figure 1.2 : A rectilinear partitioning of an attribute space.

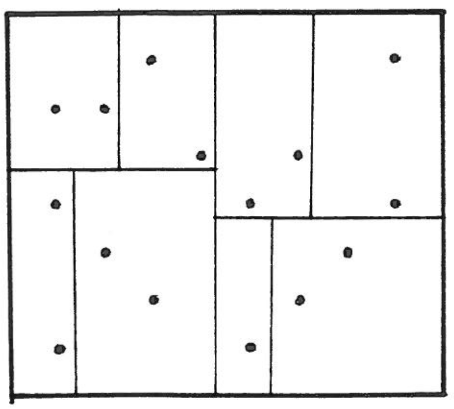

Figure 1.3 :

A non-rectilinear but rectangular partitioning of an attribute space.

multidimensional extendible hashing in section 3. In section 4, we briefly describe the concepts of uniform extendible arrays of exponential varying order and specify a storage allocation function for the elements of such an array in linear consecutive memory locations. Further details of our file organization scheme are discussed in section 5 where we present algorithms for inserting, deleting and query processing. The results of some simulation studies using both uniform and nonuniform distributed data are presented in section 6. We conclude in section 7 with some design alternatives and present suggestions for future work.

2. Review of Extendible Hashing.

We briefly review the concepts of the extendible hashing scheme of Fagin and others [9]. The reader may consult the orginal reference for detailed discussion. The file is organized in two levels: a first level of a directory, D, and a second level of a set of pages P. The directory is primarily an array whose elements are pointers to the data pages.

Each key K_i, is transformed by some encoding function to derive a pseudo-key K_i' of binary digits. The directory is indexed by choosing either the h suffix or prefix bits of K_i' and interpreting it as an integer. The value h is called the global depth of the file. The directory expands and shrinks by varying the global depth. Suppose the pseudo-key K_i' geneates the index q. Then the element D_q gives the page, P_q, where the record should reside.

Each page P_q retains a local depth h', $(h' \leq h)$, which indicates the number of prefix bits of the pseudo-keys in P_i that agree. As a result all directory elements, indexed by the pseudo-keys whose first h' bits agree point to the same page. Let the function for deriving an index into the directory, given the pseudo-key and the global depth, be denoted by g. If we take $K_i' = (\beta_0\beta_1, \ldots, \beta_{w-1})$ where w is the number of bits allowed in the binary encoding, a definition of g which allows retrievals in pseudo-key order is

$$g = \sum_{r=0}^{h-1} \beta_r * 2^r.$$

We shall assume this definition for g in any further reference to an index function. Let Ψ denote the binary encoding function and assume that $h = 2$. The Figure 2.1a shows an instantaneous configuration of the directory and data pages. To insert a key K_i, the pseudo-key $K_i' = \Psi(K_i)$, is derived. Let us assume that $K_i' =' 0111'$. Since $h = 2$, the bits $'01'$, are used as the index

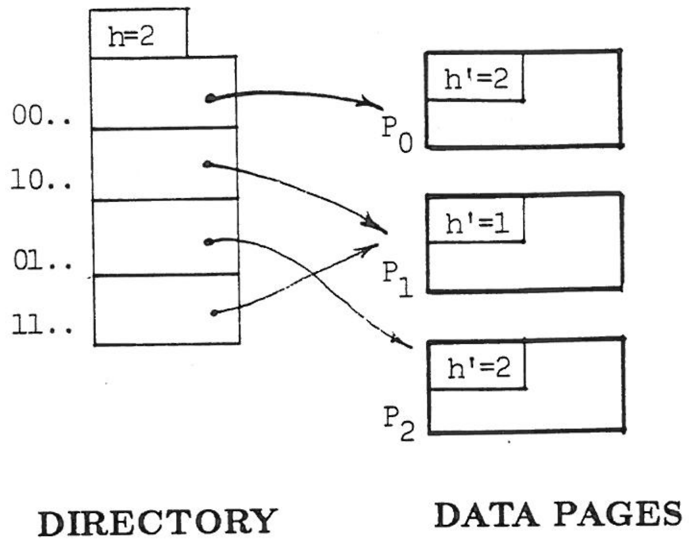

Figure 2.1a :
A schematic representation of an extendible hashing with global depth = 2.

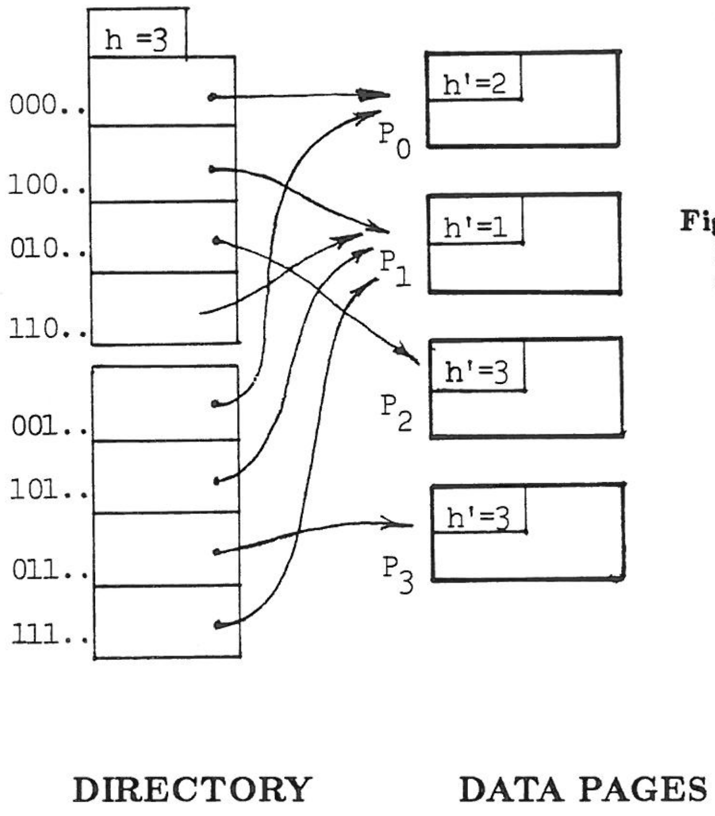

Figure 2.1b :
The configuration of the scheme after directory expansion.

into the directory. This gives the page pointer P_2 as the home page of the key K_i. As insertions continue, a page may eventually become full. At this point the local level is increased by 1 causing the keys in the page to split according to the h' bit of the pseudo-keys. One set of keys remains in the original page. The other is stored in a new allocated page. The original page is now said to be split. Occasionally such an increase in the local depth may cause it to exceed the global depth. The global depth is increased by 1, and the directory size is doubled. Suppose in inserting the key K_i into page P_2 the local depth increases to 3, then the global depth now becomes 3 and the directory must be doubled since the 3 bits can now address twice as much directory entries. The new configuration is depicted in Figure 2.1b.

In the reverse process, deletions of keys may require that two pages be merged, and when every local depth becomes less than the global depth, the directory is halved. In order to determine the point at which the directory must be halved, a count of the number of pages having a common value of the local depth h'_i are maintained for $i = 1, 2, \ldots, \lg n_d$, where $\lg X = \log_2 X$ and n_d is the directory size. In the next section we outline the generalization of extendible hashing, which is abbreviated as EXHASH, to d-dimensional keys which essentially is a d repetition of the concepts of extendible hashing one along each dimension.

3. An Overview of Multidimensional Extendible Hashing.

Suppose now that each key K_i is a vector in a d-dimensional attribute space defined by the $D_0 \times D_1 \times \ldots \times D_{d-1}$, where D_j is the domain corresponding to the j^{th} attribute, i.e., $K_i = (k_0, k_1, \ldots, k_{d-1})$ and $k_j \in D_j$. The multidimensional extendible hashing, which is abbreviated as MDEH, essentially duplicates the concept of the 1-dimensional extendible hashing for each of the d attributes, except that now, the indexes computed, for the individual attributes are used as components of a d-tuple index of a d-dimensional directory. That is for each key $(k_0, k_1, \ldots, k_{d-1})$, each component value k_j is considered to be used in a separate extendible hashing scheme. We therefore require d distinct values of each of the parameters in EXHASH. These are defined below including the definitions of some notations used latter in our discussion.

Ψ_j : The binary encoding function for j^{th} values of the key vector.

k'_j : The j^{th} pseudo-key values derived from the corresponding encoding function i.e., $k'_j = \Psi_j(k_j)$.

h_j : The global depth of the file for the j^{th} dimension.

g : A function for generating an integer index given a pseudo-key value and a gobal depth.

i_j : The index value derived from the j^{th} pseudo-key component, i.e., $i_j = g(k'_j, h_j)$.

h'_j : The local depth, for dimension j, of the file within a directory element.

n_d : The number of directory entries also refered to as the directory size.

n_p : The number of data pages allocated.

D : The directory space. A directory element will be denoted either by D_q or by $D < i_0, i_1, \ldots, i_{d-1} >$ as may be found appropriate, where $q = G_e(i_0, i_1, \ldots, i_{d-1})$ for some mapping function G_e yet to be defined.

N : The number of keys in the file.

b : The number of keys grouped per page.

δ : The number of directory elements grouped per page.

τ : A threshold value used during deletion to determine if two mergeable pages can be merged.

As in EXHASH, MDEH is organized in two levels: a first level of a directory and a second level of data pages. The directory organization constitutes the distinctive feature of MDEH. This is primarily a uniform extendible array of exponential varying order which we refer to as UXAE. A detailed discussion of such an array is presented in [22]. Other techniques for the storage allocation of extendible arrays may be found in [29]. The essential features of a UXAE as they relate to MDEH is presented in the next section. Essentially it is a rectangular array with the ability to double the range of index values along one dimension and as such it doubles the size of the array at each expansion step. The dimension which is expanded next is determined cyclically, i.e., if M is a variable that denotes the current dimension being expanded, the next dimension to be expanded is given by assigning $M \leftarrow (M + 1)$ mod d. We skip over dimensions that have exhausted the use of their bits. The reader may want to study Figures 4.1a and 4.1b to appreciate the expansion process of a UXAE.

A directory element D_q consists of a page pointer, $D_q.addr$, values for d local depths, $D_q.h'_0$, $D_q.h'_1, \ldots, D_q.h'_{d-1}$, and a variable $D_q.M$ specifying the next local depth of the file to be increased to effect a page split. We maintain the local depths in the directory unlike EXHASH where the local depth is maintained in the data pages. This enables us to avoid retaining empty pages that may result from page splitting. A vector of global depth $< h_0, h_1, \ldots, h_{d-1} >$ is maintained as a directory header such that $h_j \geq D_q.h'_j$, for j = 0, 1,...,d-1, and $q = 0,\ 1, \ldots, n_d - 1$.

To insert a key $K = (k_0, k_1, \ldots, k_{d-1})$, we generate the vector $K' = (k'_0, k_1, \ldots, k_{d-1})$ where $k'_j = \Psi(k_j)$. Next we derive for each component of the pseudo-key vector, k'_j, an index i_j where $i_j = g(k'_j, h_j)$. The function g, used as the directory index function in the EXHASH and defined in section 2, is used here to simply generate an index of a dimension. The result then is a d-dimensional vector $< i_0, i_1, \ldots, i_{d-1} >$, that addresses the directory entry D_q. Finally the destination page address is given by $D_q.addr$, where the key is stored, unless the page is found to be full in which case a page split is initiated. On the other hand $D_q.addr$ may be a null address. In this case, a new page p is allocated, the key is stored in p, and the address of p assigned to $D_q.addr$. In general a new page allocated in this manner may affect more than just one directory entry as will be realized subsequently. We assume that within a page the keys are organized by a simple sequential method.

Consider the situation where the page $p = D_q.addr$ is found to be full and must be split. The local depth $D_q.h'_m$ is increased by 1, where m $= D_q.M$. Consequently the keys in p are split into two sets according to whether bit h'_m of k'_m is 0 or 1. One set of keys are retained in the page p while the other set is assigned into a new allocated page p^*.

The values of the local depths, h'_m of all directory elements whose page pointers previously address p are updated. If $D_q.h'_m \leq h_m$, half of these page pointers are reset to point to p^*, otherwise the global depth h_m is increased by 1. The range of index values of dimension m doubles and causes the directory size to be doubled. Doubling of the directory involves the creation, for each directory element, a "buddy" in the second half. We define the concept of buddy below.

Definition 3.1

Let a directory element be addressed by $D < i_0, i_1, \ldots, i_{d-1} >$. Then the buddy of $D < i_0, i_1, \ldots, i_{d-1} >$ when the global index increases from h_m to $h_m + 1$ is $D < i_0, i_1, .., i_m + 2^{h_m}, .., i_{d-1} >$.

The MDEH organization scheme may be illustrated by inserting the following 2-dimensional pseudo-keys in the order of occurrence. {(0010, 101), (0001, 111), (0100, 000), (1111, 000), (0010, 001), (0101, 010), (0111, 100), (1010, 110), (0001, 001), (1010, 010), (0111, 111), (1101, 101), (0100, 110), (0001, 100), (0001, 100), (1111, 100) }. Starting with 4 directory entries, 4 data pages,

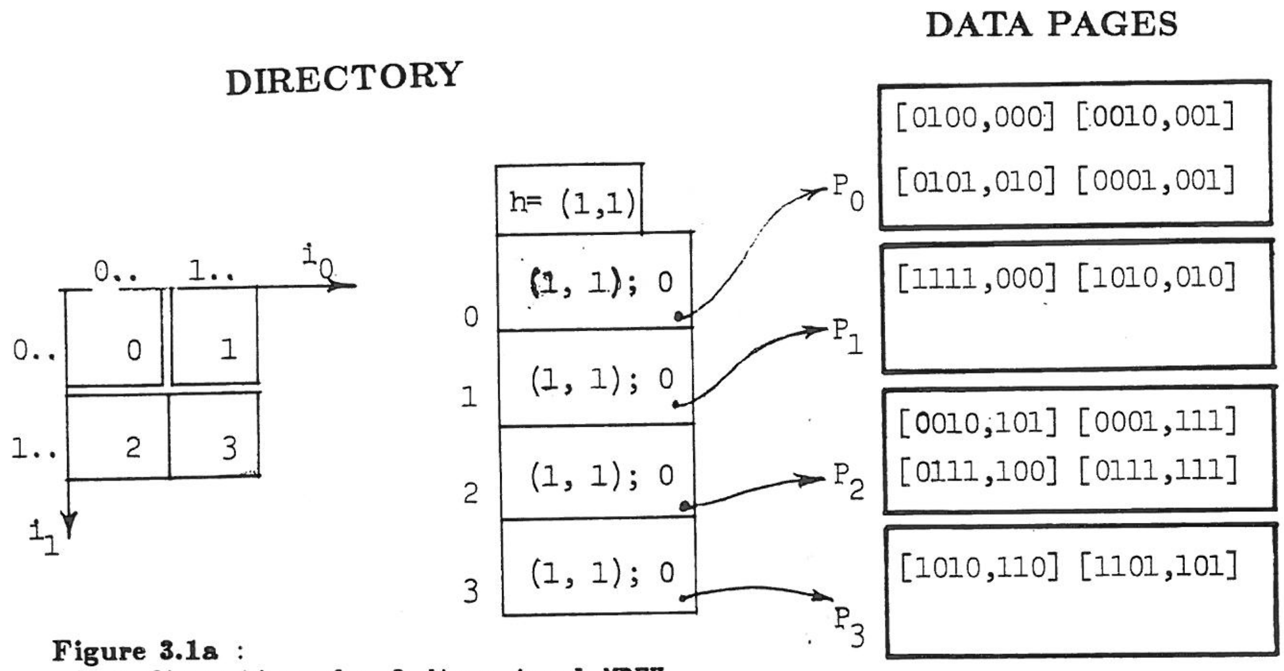

Figure 3.1a :
A configuration of a 2-dimensional MDEH with page size b = 4, after 12 key insertions.

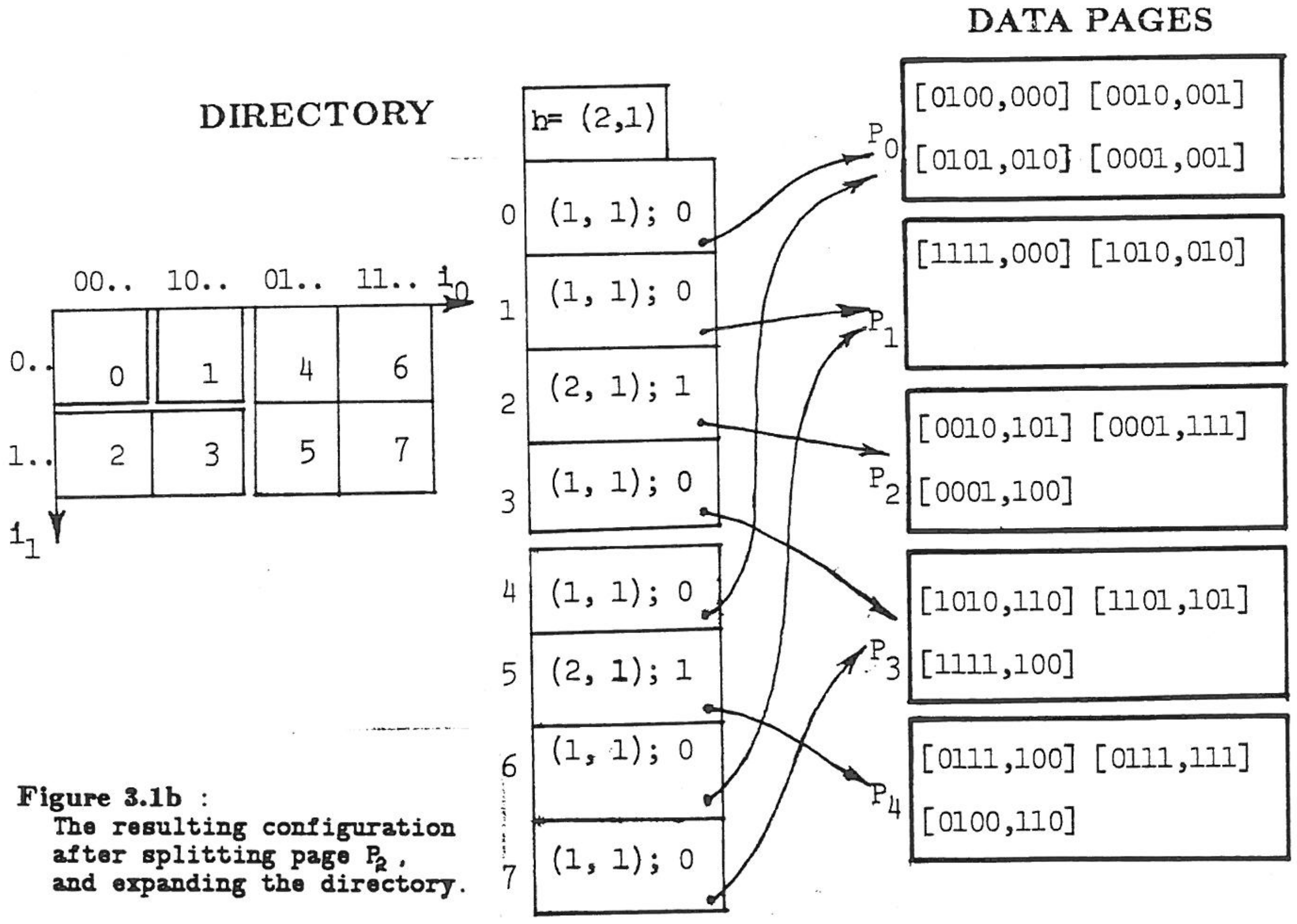

Figure 3.1b :
The resulting configuration after splitting page P_2, and expanding the directory.

Proof.

It is obvious that $G_e(0,0,\ldots,0)$ evaluates to 0 so that A<0,0,...,0> is stored in M<0>. For any arbitrary element $A < i_0, i_1, \ldots, i_{d-1} >$, each value i_j lies between 2^{h_j-1} and $2^{h_j}-1$ for some h_j except when $i_j = 0$. i.e., $2^{h_j-1} \le i_j \le 2^{h_j} - 1$. Call the block of array elements whose range of index values for dime~~s~~$\overset{ns}{}$ion j lies between 2^{h_j-1} and $2^{h_j}-1$ the adjoined block implied by i_j Every index value $i_j = 0$ implies the one element block of A<0,0,...,0>. X

From the element allocation technique, an arbitrary element $A < i_0, i_1, \ldots, i_{d-1} >$ is stored in the latest adjoined block implied by the index values $i_0, i_1, \ldots, i_{d-1}$. But this is given by i_z where z is the highest subscript such that

$$\lfloor \lg i_z \rfloor = \max(\lfloor \lg i_0 \rfloor, \lfloor \lg i_1 \rfloor, \ldots, \lfloor \lg i_{d-1} \rfloor) .$$

The size of dimension z prior to adjoining this block is 2^{h_z} where $h_z = \lfloor \lg i_z \rfloor$. The sizes J_j, j = 0,1,....,d-1, of all the dimensions prior to the expansion is given then by

$$J_j = \begin{cases} \min(2^{h_z+1}, 2^{H_j}), & \text{if } j < z; \\ \min(2^{h_z}, 2^{H_j}), & \text{if } j \ge z. \end{cases}$$

The address of the element $A < i_0, i_1, \ldots, i_{d-1} >$ is computed then as the sum of the starting address of the subarray block implied by i_z, (i.e., address of $A < 0, 0, .., i_z, .., 0 >$) and the displacement within the subarray given by the lexicographic ordering of the elements where now z is considered as the leading subscript. The address of $A < 0, 0, .., i_z, .., 0 >$ is given by $\prod_{\cancel{k}=0}^{d-1} J_r$, r X

and the displacement is given by

$$(i_z - J_z) * \prod_{\substack{j=0 \\ j \ne z}}^{d-1} J_j + \sum_{\substack{j=0 \\ j \ne z}}^{d-1} c_j i_j .$$

where $c_j = \prod_{\substack{r=j+1 \\ r \ne z}}^{d-1} J_r$. We have then that the address q of the location M<q>, into which $A < i_0, i_1, \ldots, i_{d-1} >$ is assigned, is given by

$$\begin{aligned} q \;=\; G_e(i_0, i_1, \ldots, i_{d-1}) \;&=\; \prod_{j=0}^{d-1} J_j + (i_z - \cancel{J_z}\,z) * \prod_{\substack{j=0 \\ j \ne z}}^{d-1} J_j + \sum_{\substack{j=0 \\ j \ne z}}^{d-1} c_j * i_j \; ; \\ &=\; i_z * \prod_{\substack{j=0 \\ j \ne z}}^{d-1} J_j + \sum_{\substack{j=0 \\ j \ne z}}^{d-1} c_j i_j \end{aligned}$$ X

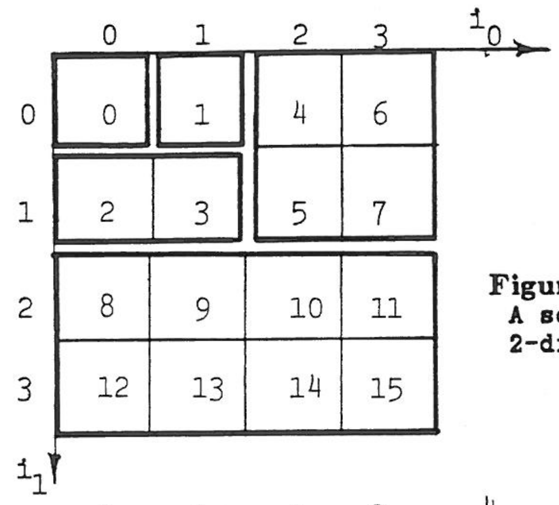

Figure 4.1a :
A schematic storage layout of a 2-dimensional UXAE A[0:3,0:3].

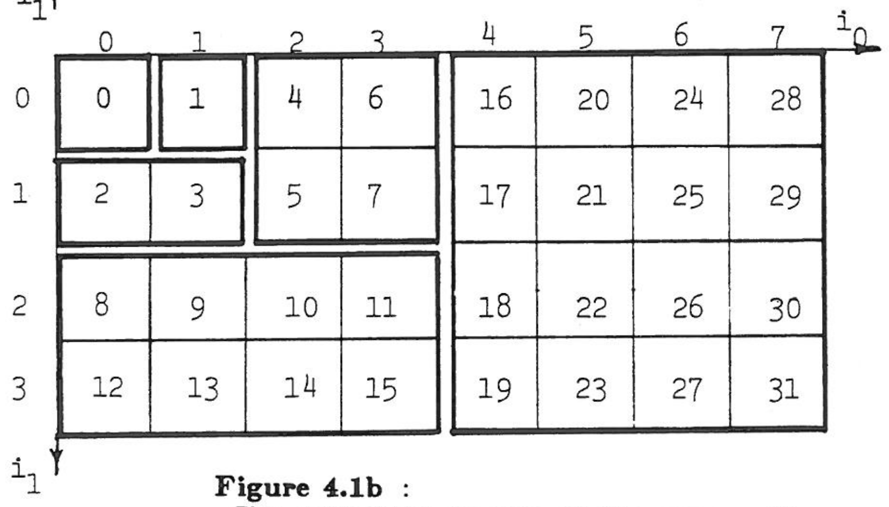

Figure 4.1b :
The schematic layout of the array after expansion to A[0:7,0:3].

This establishes Proposition 4.1.

The Figure 4.1a shows the schematic storage layout of a 2-dimesional UXAE A[0:3,0:3]. Let this be expanded to A[0:7,0:3]. Then Figure 4.1b illustrates the new storage layout obtained by adjoining the subarray which is the result of doubling the size of the first dimension. The element allocation technique for UXAE is as efficient as the storage of a d-dimensional array of fixed upper bounds since this is achieved with 100% storage utilization, and taking the logarithmic operation as an elementary operation, the time complexity to compute G_e is O(d).

From the manner in which both UXAE and the directory of MDEH expand, it is easy to appreciate why the function G_e for the UXAE defines the directory addressing function in MDEH. Recall that in section 3, we indicated that we skip over attributes that have exhausted the use of their encoded bits during the cyclic choice of the next dimension to expand. This implies, in UXAE, that the corresponding dimensions has attained its maximum allowable upper bound.

5. Insertion, Deletion and Query Processing.

5.1 Insertion.

We discuss specific consideration of the key insertion process in this section. We first address three issues that cause the number of secondary storage accesses well beyond 2 during a key insertion.

a) Whenever the directory is doubled as a result of the local depth h'_m being greater than the corresponding global depth h_m, every directory element $D < i_0, i_1, .., i_m, .., i_{d-1} >$ is duplicated as the element $D < i_0, i_1, .., i_m + 2^{h_m - 1}, ..i_{d-1} >$. The number of secondary storage accesses is equal to $2 * n_d$ where n_d is the directory size before the expansion.

b) If after the increase of h'_m, we have $h'_m \leq h_m$ and suppose the keys in page $p = D_q.addr$ (D_q being the directory element whose local depth h'_m is raised) are now distributed between p and p^* according to bit h'_m of k'_m. Then every directory element with a pointer to p must be updated. It is necessary to identify only those directory elements that must be updated. The number of directory elements affected during such a page split is given by Proposition 5.1.

c) After a page split, the keys in p, either all remain in p or may all get reassigned to the page p^* depending on whether the bit h'_m of the k'_m component of the pseudo-key vectors are either all 0 or all 1. If either p or p^* is empty, it is immediately deallocated, and all directory pointers previously pointing to it reset to null. This case automatically triggers a further page splits. Identifying the directory elements whose page pointers must be reset to null raises a special case of the problem in b).

We will first illustrate the technique to identify the directory entries to be updated in (b) and (c) before establishing the expressions for the number of storage acceses made. Consider a 2-dimensional extendible hashing scheme with global depths h_0, h_1. Suppose a pseudo-key vector is given as $K' = (\beta_0^0 \beta_1^0 \ldots \beta_{h_0}^0 \cdots, \beta_0^1 \beta_1^1 \ldots \beta_{h_1}^1 \cdots)$, where $\beta_i = \{0, 1\}$. Let K' hash to D_q, and assuming the page $p = D_q.addr$ is found to be full, with $h'_0 < h_0$, $h'_1 < h_1$ and $D_q.M = 1$. The directory entries with page pointers to p may be identified as those addressed by the pseudo-key vectors of the form

$$(\beta_0^0\beta_1^0\ldots\beta_{h'_0}^0 XXX_{h_0}\ldots,\beta_0^1\beta_1^1\ldots\beta_{h'_1}^1 0XXX_{h_1}\ldots) \text{ ....(1)}$$

and

$$(\beta_0^0\beta_1^0\ldots\beta_{h'_0}^0 XXX_{h_0}\ldots,\beta_0^1\beta_1^1\ldots\beta_{h'_1}^1 1XXX_{h_1}\ldots) \text{ ....(2)}$$

where X = {0,1} denotes "don't care" bits. The set of directory elements updated during the page split are those addressed by the pseudo-key vectors (1) and (2), where the sequences of $(h_j - h'_j)$ X's take all possible combinations of 0's and 1's. The directory elements whose page pointers are reset to point to page p^* are given by the pseudo-key vector in (2).

Proposition 5.1

Let D_q be a directory element with page pointer $D_q.addr$, and local depths $D_q.h'_j$, j = 0,1,...,d-1. Then the number of storage accesses made in splitting the page $p = D_q.addr$, with no directory expansion is $2^{\phi+1} + 4$ where $\phi = \sum_{j=0}^{d-1}(h_j - D_q.h'_j)$.

Proof.

Let the page addressed from a directory entry D_q be $p = D_q.addr$ and let the local depths in D_q be $h'_0, h'_1, \ldots, h'_{d-1}$. If the corresponding global depths are $h_0, h_1, \ldots, h_{d-1}$ then the number of directory entries whose page pointers point to p is given by

$$2^{h_0-h'_0} \times 2^{h_1-h'_1} \times \ldots \times 2^{h_{d-1}-h'_{d-1}}$$

This is equal to 2^ϕ where $\phi = \sum_{j=0}^{d-1}(h_j - h'_j)$. We require 2 accesses to locate p and detect that it is full and a further 2 accesses to secondary storage to write back the two pages resulting from the page split and $2 * 2^\phi$ directory accesses to update the affected directory entries, hence the Proposition 5.1 follows.

5.2 The Insertion Algorithm.

The algorithm to insert the key $K = (k_0, k_1, \ldots, k_{d-1})$, first checks that K is not already present. In effect, an exact-match search is first performed. We state this preliminary search phase as the algorithm FIND, for reading into memory, the destination page of the key. The algorithm FIND returns true if the key is already present otherwise it returns false. The insert algorithm references an array called COUNT whose significance will be apparent when we discuss deletion. We shall assume also that every attribute has the same numebr of bits in the encoding.

FIND (Given the key $K = (k_0, k_1, \ldots, k_{d-1})$) ;

F1 : Compute $k'_j \leftarrow \Psi_j(k_j)$; $i_j \leftarrow g(k'_j, h_j)$, j = 0,1,...,d-1;

F2 : Compute $q \leftarrow G_e(i_0, i_1, \ldots, i_{d-1})$;

F3 : Access the directory element D_q; Set $p \leftarrow D_q.addr$;

F4 : If $p \neq$ null then access page p and if K is in p return "true" else return "false".

INSERT (Given the key $K = (k_0, k_1, \ldots, k_{d-1})$) ;

I1 : If **FIND**(K) then return "error message",

I2 : If p is null allocate a new page p^*, set $p \leftarrow p^*$, store K in p and goto I3. If page p is not full then store K in p and return. Allocate a new page p^*; set $m \leftarrow D_q.M$; if $D_q.h'_m < h_m$ goto I3. For each d-tuple index $< i_0, i_1, \ldots, i_{d-1} >$ and for all combinations of values of i_j, $0 \leq i_j \leq 2^{h_j} - 1$, j = 0,1,..., d-1, compute $r \leftarrow G_e(i_0, \ldots, i_m, \ldots, i_{d-1})$; $t \leftarrow G_e(i_0, \ldots, i_m + 2^{h_m}, \ldots, i_{d-1})$ and assign $D_t \leftarrow D_r$. Set $h_m \leftarrow h_m + 1$; $n_d \leftarrow 2 * n_d$.

I3 : Compute $k'_j \leftarrow \Psi_j(k_j)$, set $v_j \leftarrow 0$ for j = 0,1,...,d-1; $x \leftarrow \sum_{j=0}^{d-1} D_q.h'_j$, $COUNT[x] \leftarrow COUNT[x] + 1$.

I4 : Let $h'_j = D_q.h'_j$ for j = 0, 1,...,d-1. Insert the $(h_j - h'_j)$-bit binary representation of v_j as the bits h'_j to h_j of k'_j and compute $i_j \leftarrow g(k'_j, h_j)$ for $j = 0, 1, \ldots, d-1$. Set $r \leftarrow G_e(i_0, i_1, \ldots, i_{d-1})$; $D_r.h'_m \leftarrow D_r.h'_m + 1$; $D_r.M \leftarrow (D_r.M + 1)$ mod d. If bit h'_m of k'_m is 1 set $D_r.addr \leftarrow p$. Set $j \leftarrow 0$.

I5 : Set $v_j \leftarrow v_j + 1$; if $v_j < 2^{h_j - h'_j}$ goto I4 otherwise set $v_j \leftarrow 0$; $j \leftarrow j + 1$ and if j < d repeat I5.

I6 : Store the keys in p into a temporary area Q and clear page p. For each key K in Q do **INSERT**(K).

I7 : If either page p or p^* is empty, reset all pointers in the directory that point to it to null, and deallocate the page. Set $COUNT[x] \leftarrow COUNT[x] - 1$;

I8 : **INSERT**(K);

Note that the recursive call of INSERT at I6 can never startup further page split while that at (I8) may initiate subsequent splits, and possibly a directory expansion.

5.3 The Deletion Algorithm.

The deletion of a key $K = (k_0, k_1, \ldots, k_{d-1})$ essentially involves a FIND to locate the page where the key is resident and then freeing the space occupied by the key. Two problems need to be considered after a delete operation.

i. To retain a high load factor, two pages addressed by directory elements that are buddies must be merged if this can be done without incurring overflows.

ii. The directory size must be halved whenever every directory element points to the same page as its sibling.

To resolve (i), a threshold value τ defined as the the fraction of the page capacity that must be utilized is specified. Suppose a key is deleted from page p. Let D_q be the directory element whose page pointer addresses p, and let D_{q^*} be the buddy of D_q where $p^* = D_{q^*}.addr$. Then p^* is merged into p if the sum of their contents is less than $\tau * b$, where b is the page capacity.

Detecting when the directory must be halved to resolve the problem in (ii) is done with the aid of a 1-dimensional array COUNT[0..(lg n_d)], (n_d is the directory size). The entries of the COUNT array are the number of data pages that are addressed by directory elements with a common value of the local depth. The sum of the local depths serves as the index into the array COUNT. The directory is halved whenever $COUNT[\lg n_d] = 0$. We state the algorithm for deletion as follows. We shall assume, as in INSERT, that we do not skip over any attributes.

DELETE (Given key $K = (k_0, k_1, \ldots, k_{d-1})$) ;

D1 : If not FIND(K) then return "error message".

D2: Delete the key by freeing the space occupied by K in p.

D3: Compute $k'_j \leftarrow \Psi(k_j)$ for $j = 0, 1, \ldots, d-1$. Set $m \leftarrow (D_q.M + d - 1)$ mod d. Reverse the bit h'_m of k'_m.

D4 : Compute $i_j \leftarrow g(k'_j, h_j), j = 0, 1, \ldots, d-1$; and set $q^* \leftarrow G_e(i_0, i_1, \ldots, i_{d-1})$. Access page $p^* = D_{q^*}.addr$.

D5 : If the sum of the keys in p and p^* is greater than $r * b$ then return. {If D_q is addressed using $(k'_0, k'_1, \ldots, k'_{d-1})$ such that bit h'_m of k'_m is 0, reverse the role of D_{q^*} and D_q as well as p^* and p in D6 - D9.}

D6 : Assign the keys in p^* to p; and set $v_j \leftarrow 0$, j = 0, 1,...,d-1. Set $x \leftarrow \sum_{j=0}^{d-1} D_{q^*}.h'_j$; $COUNT[x] \leftarrow COUNT[x] - 1$.

D7 : Insert the $(h_j - h'_j)$-bit binary representation of v_j as the bits h'_j to h_j of k'_j and compute $i_j \leftarrow g(k'_j, h_j)$ for $j = 0, 1, \ldots, d-1$. Set $r \leftarrow G_e(i_0, i_1, \ldots, i_{d-1})$, $D_r.h'_m \leftarrow D_r.h'_m - 1$, $D_r.M \leftarrow (D_r.M + d - 1)$ mod d. If bit h'_m of k'_m is 1, set $D_r.addr \leftarrow p$. Set $j \leftarrow 0$.

D8 : Set $v_j \leftarrow v_j + 1$; if $v_j < 2^{h_j - h'_j}$ goto D7 otherwise set $v_j \leftarrow 0; j \leftarrow j + 1$; and if j < d repeat D8.

D9 : If $n_d > 1$ then if $COUNT[\lg n_d] = 0$, halve the range of index values of dimension $D_q.M$ and free the space of the lower half of the directory elements; $n_d \leftarrow n_d/2$. Return.

5.4 Query Processing.

We consider the algorithm for processing partial-match seach in this section. The algorithm to process an exact-match search is equivalent to the FIND algorithm specified in the preceding section. Where the binary encoding function is order preserving, the partial-match algorithm, yet to be stated, is easily adapted for processing range and partial-range queries. Precise definitions of the various query classes may be found in [1, 3, 22, 27, 31]. A partial-match query requires the retrieval of all keys that match some s out of d specified combination of attribute values, where $s \leq d$. Let an unspecified value be denoted by asterisk ('*'). The algorithm Partial-Match given below retrieves all pages containing keys whose s corresponding attributes match the specified attribute values.

Partial-Match (Given $k_0, k_j, \ldots, k_{d-1}$, of which d-s of them are asterisks)

PM1 : For each value k_j, j = 0, 1,...,d-1 do the following. If $k_j \neq ' *'$ then $l_j \leftarrow u_j \leftarrow g(\Psi(k_j), h_j)$; else set $l_j \leftarrow 0, u_j \leftarrow 2^{h_j}$.

PM2 : Set $i_j \leftarrow l_j$, j = 0, 1,...,d-1;

PM3 : Compute $q \leftarrow G_e(i_0, i_1, \ldots, i_{d-1})$; Access D_q; $p \leftarrow D_q.addr$; If page p has not been read then read page p and retrieve all keys K in p, whose corresponding s components match the specified s values. Set $j \leftarrow 0$.

PM4 : Set $i_j \leftarrow i_j + 1$; if $i_j < u_j$ goto PM3, otherwise set $i_j \leftarrow l_j$, $j \leftarrow j + 1$; and if j < d repeat PM4 else return.

Let s be the cardinality of the set S of the specified attributes of the key. Examination of the steps of the partial-match algorithm above reveals that the number of directory elements is equal to (d-s)-dimensional block of size $n_d / \prod_{j \in S} 2^{h_j}$. The number of actual data page accesses is less than or equal to this number since in the worst case there would be one page for each directory element. If the encoding functions for the various attributes are order preserving, the partial-match algorithm may be modified to handle range queries in an obvious way by appropriate initialization of the l_j's and the u_j's.

6. Simulation.

Three performance criteria for a data organization system are the load factor (α), the average number of secondary storage accesses required for an exact-match search (π_e), and the average number of secondary storage accesses made to insert a key (ρ). We describe some simulation experiments done to estimate these measures of performance. The objectives of the experiments are primarily to

a) assess the average load factor with increasing file size.

b) assess the average number of disk accesses for a successful exact-match search when I/O buffering is used.

c) assess the average number of disk reads reqiured to insert a key.

d) assess the directory size with the file growth.

The estimates (a) to (d) above were done with varying page sizes and for two different data distributions in the attribute space; i) Independent Uniform distribution, and ii) Non-uniform distribution using a Multivariate Normal. We present results for page sizes of 20 and 50 only and for the dimensionality d = 2, since our experiments show no significant changes when d is either 3 or 4.

6.2 The Simulation Model.

The two distribution of keys used in the experiments are derived as follows.

Uniform distribution (Case I).

The attribute values k_j, j = 0, 1,...,d-1, are generated independently of each other and uniform between 0 and MAX, where MAX is the maximum integer in a 32 bit word.

Multivariate Normal (Case II).

The values of the k'_js are derived by discretizing from a multivariate normal distribution with a covariance matrix of

$$\begin{bmatrix} 1.0 & 0.8 \\ 0.8 & 1.0 \end{bmatrix}$$

The spatial patterns of the two distributions are shown in the Figures 6.1 and 6.2.

We assume a paged memory environment with 16 page frames. A data page has capacity for b keys while a directory page has capacity for δ directory elements. The results presented are for $\delta = 63$ and b = 20, 50. The least recently used (LRU) page replacement policy is enforced. We observe the parameters α, π_e, ρ and n_d for every 200 insertions as the file grows from N = 1 to N= 10000 keys, both for Case I and Case II.

6.3 Simulation Results.

The graphs of the variation of the average number of actual disk accesses required to perform a successful exact-match search with increasing file size is shown in Figure 6.3. Even with buffering, the parameter π_e eventually attains the value of 2. The value of 2 is attained faster with non-uniform distributed key values. This behaviour is attributed to the fact that Case II sustains an initial faster rate of growth of the directory which soon results in a page fault occuring each time the directory is accessed. There is little effect on the parameter π_e in Case II when the page size is increased. However, in the case of uniform distributed values, the values of π_e significantly decreases when the page size is increased from 20 to 50.

The load factor values observed for page sizes of 20 and 50 as the file size increases are shown in Figure 6.4. The load factor values for uniform distributed keys oscillates between 0.65 and 0.74 for a page size of 20. The amplitude of oscillation is about 0.1 for a page size of 50. The average value is about 0.7. For Case II, the load factor values show gradual increasing values with little fluctuations, which eventully settle at about 0.67 for the two different page sizes. It appears that

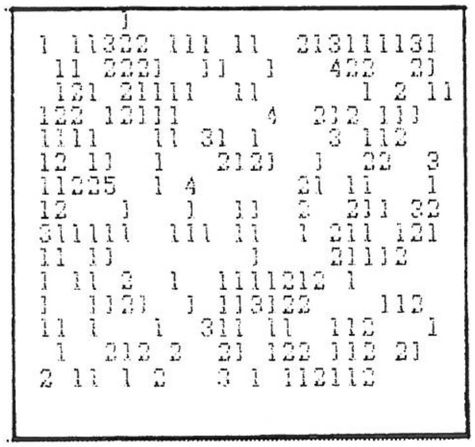

Figure 6.1 :
The spatial pattern of a 2-dimensional uniform distributed keys.

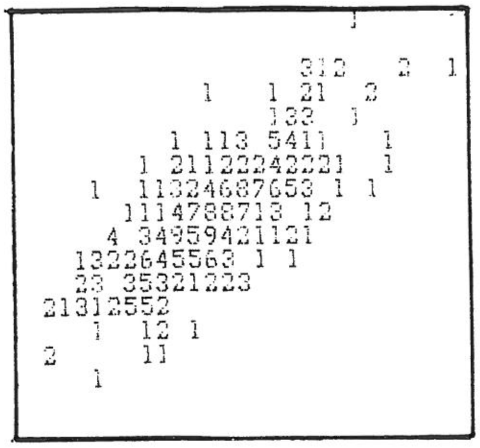

Figure 6.2 :
The spatial pattern of a 2-dimensional multivariate normal distributed keys.

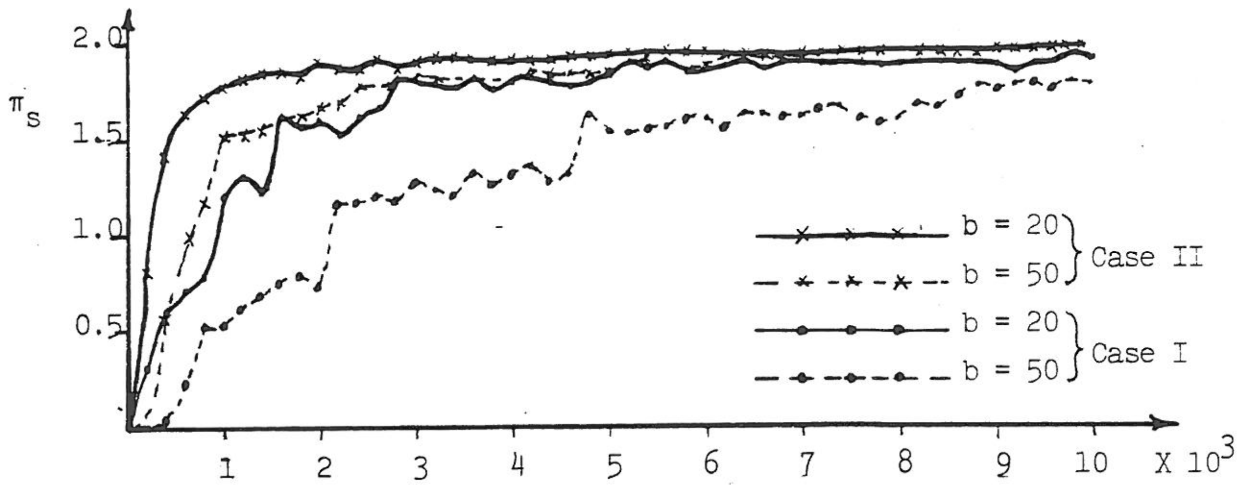

Figure 6.3 : **The average number disk reads for a successful exact-match search.**

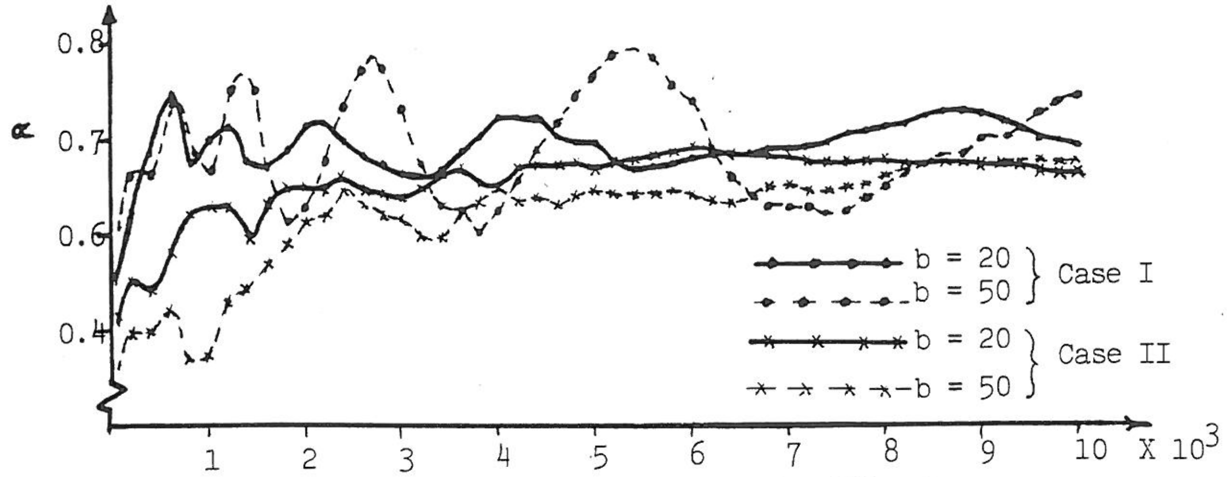

Figure 6.4 : **The average load factor with increasing file size.**

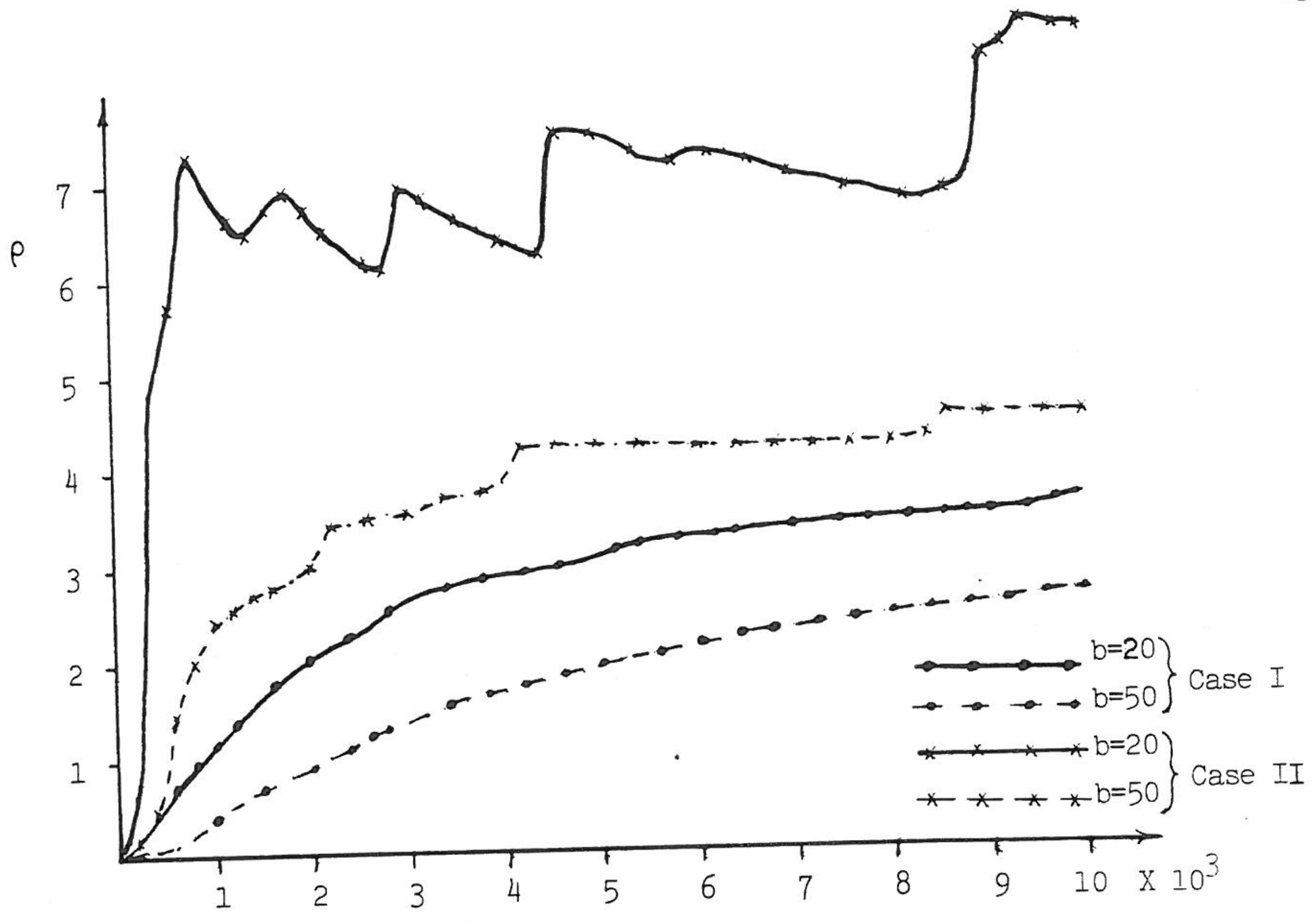

Figure 6.5 : **The average number of disk reads per insertion.**

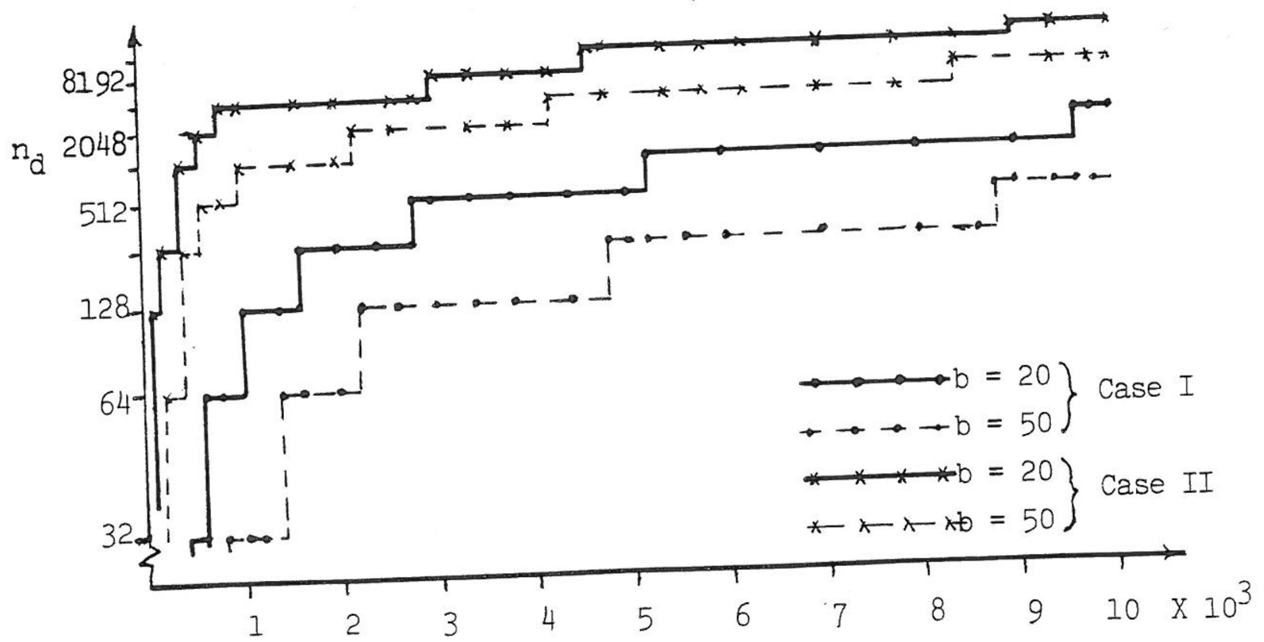

Figure 6.6 : **A log-normal plot of the variation of the directory size with the file size.**

irrespective of the distribution and the page sizes, the average load factor value is eventually about 0.69. This seemingly asymtotic average value of 0.69 has been observed in similar page splitting file design schemes in [2, 16, 20, 24].

Figure 6.5 shows the graphs of the average number of actual disk accesses required per key insertion. The variation of ρ for keys uniformly distributed in the key space is not only significantly lower than the values for non-uniform distributed keys but also shows a steady smooth slow growth unlike Case II which shows a more oscillatory variation. Such a behaviour is expected of the multivariate normal distribution since the keys hash mainly into the directory positions corresponding to the cells along the diagonal. The result is a high rate of collision and consequently a high rate of page splitting and a relatively large directory size particularly for small page sizes.

The Figure 6.6 depicts the growth of the directory size. The values of n_d, are shown on a logarithmic scale. The graphs exhibit a staircase-like variation as the file grows. The jump points represent the instants at which the directory size doubles. For the same number of key insertions and the same page size, the directory sizes in Case II are about 16 to 32 times the values in Case I. This highlights the importance of using a randomizing function for the binary encoding of the key values to achieve almost uniform distribution of the pseudo-keys. Some randomizing techniques are discussed in [6,11].

7. Discussions and Design Variants.

A number of noteworthy features have been demonstrated by our preliminary simulation results.

1. Even with buffering, the average number of page accesses to locate a key soon reaches the upper bound of 2.
2. The load factor or occupancy ratio α, fluctuates about a mean value of 0.69 and is independent of the data distribution and the page size.
3. To maintain a relatively small directory size, the encoding functions should be a good randomizing function to generate components of the pseudo-keys that are uniform and independent. Unfortunately the scheme under such conditions looses its order preserving property and consequently its ability for efficient range searching.

Our presentation has been based on a splitting policy that assumes that each attribute in the data space is equally likely to be specified in a query. This is not generally true in practice. Where the probability of an attribute participation in a query is known, a design objective will be to minimize the number of directory and page accesses for a partial-match search. This issue is discussed in [16,25]. Essentially, a derived vector called a "choice vector" dictates the dimension to be expanded next. The use of such a "choice vector" is easily accommodated in MDEH by a slight redefinition of the mapping function G_e and retaining the starting element address of each adjoined subdirectory block at each expansion step. See [19,21,22] on the use of such techniques. This requires an extra storage of size $O(\lg n_d)$ and a time complexity of $O(\lg n_d)$ to compute G_e.

A problem related to the growth of the directory size is the expected number of prefixed bits used per attribute given N key insertions. The problem has been independently addressed by Devroye [8] and Regnier [26]. Assuming that the attribute values are independent and the keys are grouped b per page, it is our conjecture that if $S_h = \sum_{j=0}^{d-1} h_j$, is the sum of global depths of the file after N keys insertions, then the expected value of S_h is

$$E(S_h) = (1 - 1/b) \ * \ lg \ n_d \ ;$$

where n_d is the directory size. We address this problem and other related analytical results in a follow up paper. Issues such as the concurrent manipulation of such structures are being studied and an implementation of the scheme for use in practice is currently under way at Carleton University in the design of an integrated data organization method for associative searching.

Acknowledgement.

This research is supported in part by grants from the National Research Council of Canada and GR5 grant from Carleton University. I would also like to thank N. Santoro for his helpful comments and Nel Holtz for facilitating the typing of this paper.

References.

[1] Aho A. V. and Ullman J. D. Optimal Partial-Match Retrieval When Fields are Independently Specified. *ACM Trans. Database Syst., 4, 2 (Jan 1979) 168–179.*

[2] Bayer R. and McCreight E. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica 1, 3 (1972), 173-189.*

[3] Bentley J. L. Multidimensional Binary Search Tree Used For Associative searching. *Comm. ACM 18, 9 (Sep. 1975), 509–517.*

[4] Bolour A. Optimality Properties of Multiple Key Hashing Functions. *J. ACM 26, 2 (Apr. 1979), 196–210 .*

[5] Burkhard W. A. Interpolation-Based Index Maintenance. *Proc. 2nd Symp. on Principles of Database Systems (1983), 76–88 .*

[6] Carter J. L. and Wegman M. N. A Universal Class of Hash Functions. *Journal of Comput. and Syst. Science, 18, (1979), 143–154 .*

[7] Chang C. C., Lee R. C. T. and Du H. C. Some Properties of Cartesian Product Files. *ACM SIGMOD Conf., Santa Monica, (May 1980), 157-168.*

[8] Devroye L. A Probabilistic Analysis of the Height of Tries and the Complexity of Trie Sort. *(To Appear).*

[9] Fagin R., Nievergelt J., Pippenger N. and Strong H. R. Extendible Hashing: A Fast Access Method for Dynamic Files. *ACM. Trans. Database Syst., 4, 3 (Sep. 1979), 315-344.*

[10] Finkel R. A. and Bentley J. L. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica, 4, 1 (1974), 1-9.*

[11] Knuth D. E. The Art of Computer Programming. Vol. 3: Sorting and Searching. *Addison Wesley, Reading Mass. 1973.*

[12] Larson P. A. Dynamic Hashing. *Bit, 18, 2 (1978), 184-201.*

[13] Larson P. A. Linear Hashing With Partial Expansion. *Proc. 6th intern. Conf. VLDB, Montreal (1980), 224–232.*

[14] Liou J. H. and Yao S. B. Multidimensional Clustering for Database Organization. Inform. Syst., 2, 4 (1977), 187–198.

[15] Litwin W. Linear Hashing: A New Tool for File and Table Addressing. *Proc. 6th Intern. Conf. VLDB, Montreal (1980), 212–223.*

[16] Lloyd J. W. and Ramamohanarao K. Partial-Match Retrieval for Dynamic Files. *Bit, 22 (1982), 150-168.*

[17] Lomet D. Bounded Index Exponential Hashing. *ACM. Trans. Database Syst., 8, 1, (Mar. 1983), 136-165.*

[18] Lomet D. A High Performance Universal, Key Associative Access Method. *Proc. ACM SIGMOD Conf. (1983), 120–133.*

[19] Merrett T. H. and Otoo E. J. Dynamic Multipaging: A Storage Structure for Large Shared Data banks. *Proc. 2nd Intern. Conf. on Databases. Improving Usability and Responsiveness, Academic Press (1982), 237–256.*

[20] Nievergelt J., Hinterbeger H. and Sevcik K. C. The Grid File: An Adaptatable Symmetric Multi-Key File Structure. *Eidgenössische Technishe Hochschule Zürich. Institut für Informatik (Dec. 1981).*

[21] Otoo E. J. and Merrett T. H. A Storage Scheme for Extendible Arrays. *Computing 31 (1983), 1–9.*

[22] Otoo E. J. Low Level Structures in the Implementation of the Relational Algebra. *Ph. D. Thesis, School of Computer Science, McGill University (Aug. 1983).*

[23] Otoo Ekow J. Dynamic Multidimensional Hashing for Files With Composite Keys. *Technical Report (1984), School of Computer Science, Carleton University, (Submitted).*

[24] Ouksel M. and Scheuermann P. Storage Mapping for Multidimensional Linear Dynamic Hashing. *Proc. 2nd Symp. on Principles of Database Systems, Atlanta, Georgia (1983), 90–105.*

[25] Pfaltz J. L., Berman W. J. and Cagley E. M. Partial-Match Retrieval Using Indexed Descriptor Files. *Comm. ACM, 23, 9 (Sep. 1980), 522–528.*

[26] Regnier M. On The Average Height of Tries in Digital Search and Dynamic Hashing. *Inform. Proc. Lett., 13, 2 (Nov. 1981) 64–65.*

[27] Rivest R. L. Partial-Match Retrieval Algorithms. *SIAM Journal of Comput., 5, 1 (Apr. 1976), 19–50.*

[28] Robinson J. T. The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes. *ACM SIGMOD Conf. Ann-Abor (Apr. 1981), 10–18.*

[29] Rosenberg, A. L. Managing Storage for Extendible Arrays. *SIAM J. Comput. 4, 3 (Sep. 1975), 287–306.*

[30] Rothnie J. B. and Lozano T. Attribute Based File Organization in Paged Memory Environment. *Comm. ACM, 17, 2 (Feb. 1974), 63–69.*

[31] Scheuermann P. and Ouksel M. Multidimensional B-Trees for Associative Searching in Database Systems. *Inform. Syst., 7, 2 (1982), 123–137.*

[32] Tamminen M. The EXCELL Method for Efficient Geometric Access to Data. *Acta Polytechnica Scandinavica. Mathematics and Computer Science Series No 34, (1981). CAD Project, Helsinki, University of Technology, Laboratory of Information Processing Science.*