COMPUTED ACCESS METHODS FOR EXTENDIBLE ARRAYS OF VARYING DIMENSIONS

Ekow J. Otoo

SCS-TR-58

August 1984

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

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

Computed Access Methods for Extendible Arrays of Varying Dimensions.

Ekow J. Otoo

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

Abstract

We introduce a method of managing storage for dense rectangular arrays in consecutive memory locations such that the number of dimensions as well as the index range of each dimension can be extendible. The method realizes an n-element array of any shape in $\Theta(n)$ storage locations with a computed access function of complexity $O(d^2)$, where n and d are respectively, the size and dimensionality of the array at any instant. This presents a solution to a more difficult variant of an open question of Rosenberg that asks whether or not a realization exists which favours non-trivial infinite family of shapes such that an array of n elements is spread over at most $C * n$ locations for some integer C. We show that $C \leq 3.25$ for any d and $n >> d$.

Keywords and phrases : Data structures, extendible arrays, storage allocation, mapping functions, storage utilization.

CR Categories and Subject Descriptors : E.1 [Data Structures] : arrays; E.2 [Data Storage Representation]; H.2.2 [Database Management] : Physical Design; H.3.3 [Information Storage and Retrieval] : Information Search and Retrieval.

1. Introduction

Computed access techniques for mapping elements of dense rectangular arrays of bounded size and fixed number of dimensions in consecutive storage locations have the virtue of fast element access and efficient storage utilization. Unfortunately they lack the flexibility of admitting extensions in the size of the dimensions except for one. Furthermore, the array must be defined to be of a fixed number of dimensions that cannot increase during the life time of the array. Considering that the array is a fundamental data structure with wide application in engineering and mathematical sciences, relatively little work is reported on mapping functions for extendible arrays. There is no known strategy that uses computed access method for allocating and retrieving elements of arrays that, not only allows extension of the range of subscript values, but also allows the number of dimensions to grow. This paper addresses the problem of storing such dense rectangular extendible arrays.

Let N be the set of non-negative integer. A declaration of an array is of the form $A[l_1 : u_1,\ l_2 : u_2, \ldots, l_d : u_d]$, where $l_j,\ u_j \in N$, and $j = 1, 2, \ldots, d$. An element of the array is referred to as $A\langle i_1, i_2, \ldots, i_d\rangle$, for $l_j \leq i_j \leq u_j$. An array is realized in storage as a set of consecutive memory locations $M[0 : n-1]$, where $n = (\prod_{j=1}^{d} (u_j - l_j + 1))$. Suppose an element $A\langle i_1, i_2, \ldots, i_d\rangle$ is stored in location $M\langle q\rangle$ then the index q is given by a mapping function $\Re : N^d \rightarrow N$, such that $q = \Re(\langle i_1, i_2, \ldots, i_d\rangle)$. The mapping function $\Re$ maps the d-tuple indices one-to-one onto the n consecutive locations. This method of realizing arrays is referred to as a computed access method (see [12]). Typical computed access methods for bounded rectangular arrays, are exemplified by the conventional row-major (or lexicographic) order and the column-major order of element allocations.

A drawback in such conventional techniques, such as the row-major order, is that it allows the expansion of the array only along one dimension. Rosenberg [13] refers to arrays realized in this way as "prism arrays".

Consider first the case when the array is of fixed number of dimensions. One can envisage a scenerio in which the upper bounds of the range of index values of each dimension may be progressively increased on demand. Assuming a d-dimensional array is initially declared as $A[l_1 : u_1, \ldots, l_j : u_j, \ldots, l_d : u_d]$. Then at some point in time, the dimension j may be extended so that the new array becomes $A[l_1 : u_1, \ldots, l_j : u_j + 1, \ldots, l_d : u_d]$. The

effect of the expansion is to admit a (d-1)-dimensional subarray of size $\prod_{\substack{r=1 \\ r \neq j}}^{d} (u_r - l_r + 1)$ elements. A computed access method for realizing extendible arrays requires that the new set of elements be allocated in conservative locations following the block of locations occupied by the previous array and that the mapping function correctly addresses every element of the array without redefinition or storage reorganization.

1.1 Related Works.

Pioneering works in techniques for realizing extendible arrays of fixed dimensionality are presented in a series of papers by Hansson and Tänlund [2], Rosenberg [12, 13], Stockmeyer [16] and also jointly by Rosenberg and Stockmeyer [14]. Their works investigated extendible array realizations using four criteria for measuring the quality of a realization, viz., simplicity of arbitrary element access, simplicity of traversal along a path, simplicity of extension and efficiency of storage utilization.

It is shown in [13] that the lower bound on the storage requirement, for an extendible array of n elements, for a storage mapping with unbridled extendibility is $O(n\,(lg\,n)^{d-1})$. However, by restricting the pattern of expansion of the array and allowing only arrays that conform to the restriction, the storage requirement is improved to $O(n)$. Two methods of storing n-element extendible arrays with restricted pattern of growth in $O(n)$ locations are demonstrated by the diagonal and cubical shell index realizations [12,13].

Hansson and Tärnlund [2] present another approach for realizing arrays having a prescribed pattern of growth. The technique called the "Zig-zag procedure" achieves 100% storage efficiency since the array is constrained to be always cubical at each expansion step.

Similar storage efficient realizations for arrays with restricted pattern of growth have been demonstrated by our earlier works on storing extendible arrays whose range of subscript values can be extended either linearly or exponentially (see [6, 7,8]). Ouksel and Scheuermann [11] have presented a method of representing cubical arrays having linear expansions, such that an n element array is represented in n locations. The mapping function however, is computable in $O(log\,n)$ which is not independent of the size of the array.

In [13] the question was asked regarding the lower bound on the storage utilization of extendible array realization that admit infinitely may shapes. In particular, the author asks

whether or not a realization exists which favours, non-trivially, infinite family of shapes such that an array of n element is spread over $C*n$ locations. The paper further reported a suggestion by Shmuel Winograd who asked whether added information might allow one to overcome the $O(n\,lg\,n)$ lower bound. We have shown in an earlier paper [10], in introducing the "Index-Array" technique for extendible array, that the answer is affirmative. However the earlier discussed method does not admit extensions in the number of dimensions.

The Index-Array technique allocates a d-dimensional array of n elements using $\Omega(n)$ locations in the best case and $O(n*d^2)$ locations in the worst case. The complexity of the mapping function in the Index-Array scheme is $\Theta(d)$. Essentially the Index-Array scheme computes the address of an element $A\langle i_1, i_2, \ldots, i_d\rangle$ using a row-major order function of the form

$$\Re(\langle i_1, i_2, \ldots, i_d\rangle) = s_0 + c_1\,i_1 + c_2\,i_2 + \ldots + c_d\,i_d\,.$$

for any d-tuple index. The value of s_0 is the starting address of some block of elements in which $A\langle i_1, i_2, \ldots, i_d\rangle$ belongs. The values $s_o, c_1, c_2, \ldots, c_d$ are information stored in locations given by a 3-dimensional array called the Index-Array. This added information enables the Index-Array scheme to overcome the $O(n\,(lg\,n)^{d-1})$ lower bound as rightly suggested by Winograd.

Linked allocation techniques, such as the use of vector of pointers, linked list, orthogonal list (see [4,15]) and even hashing techniques [14] may be used in array representations that admit extendibility. What then makes computed access methods attractive ?

The answer is in the advantages that may be derived and in the application of the realization functions to other data organization schemes. Two advantages of computed access realization are stated in [13], viz., i) such array realizations afford one both easy probing of the array, which are lacking in linked schemes, and ii) they allow easy traversal along, say, rows and columns of the array, which is not present in hashing schemes. Issues such as proximity of array elements and ease of traversal are discussed in [1] and [14].

A third reason for the quest of computed access techniques is that the mapping functions that are derived have applications in multidimensional dynamic hashing schemes for physical data organization. This is evident in the multidimensional storage schemes presented in [6, 7, 8, 9, 11]. The use of the mapping functions for extendible arrays to databases constitutes the primary motivation for our research.

The earlier works on extendible arrays have only considered the case where the number of dimensions is kept fixed, but the range of subscript values gradually increases. We term expansions of this type "longitudinal". An array that has the propensity to increase the number of dimension is said to admit "lateral" expansion. We are concerned in this paper with methods of managing storage for extendible array that allow both longitudinal and lateral expansions.

1.2 Motivation.

The current studies has been influenced by our earlier works on storage mapping functions for multidimensional dynamic hashing schemes [6, 7, 8, 9]. Essentially, our quest for a page addressing function that is appropriate for the Dynamic Multipaging scheme [6, 9], led to the Index-Array scheme for extendible arrays.

The idea of the dynamic multipaging scheme is to store a file of d-attribute records on secondary storage in pages (or blocks) addressed from 0 to $n-1$ say. Each field v_j of a record $r_i = \langle v_1, v_2, \ldots, v_d \rangle$, is used to generate an integer value i_j. The d integer numbers derived in this way, are used to form a d-tuple index $\langle i_1, i_2, \ldots, i_d \rangle$. Given a suitable mapping function $\Re_\alpha$, the page address p , is computed as $p = \Re_\alpha(\langle i_1, i_2, \ldots, i_d \rangle)$ The value of n increases as pages are added to accommodate inserted records and decreases as pages a deleted to reclaim the space after record deletions. As we have shown in [9], the page addressing functions $\Re_\alpha$ is defined by the the mapping function for extendible arrays.

Suppose that a file of substantially large number of records is organized using dynamic multipaging. Some justifiable reason might dictate, at some point in time, that one or more fields be added to all subsequent records inserted into the file. If the added field is also a search field of the record, then as in all multidimensional structures, the whole file must either be reorganized to take account of the new fields, or the page addressing function must be redefined with exceptional condition statements. The dynamic multipaging scheme then does not posses the ability to expand laterally. To instill both longitudinal and lateral expansion capability in dynamic multipaging, the function $\Re_\alpha$ must be defined as an equivalent mapping function for extendible arrays of varying dimensions. We address the problem of defining such a mapping function in this paper, only in the context of realizing extendible arrays.

1.3 Problem Formulation.

Let N be the set of non-negative integers. Consider an array $A_\alpha[0:u_1, 0:u_2, \ldots, 0:u_d]$ where $d, u_j \in N$ for j = 1,2,...,d. The values $d, u_1, u_2, \ldots, u_d$, signify the number of dimensions and the upper bounds on the range of subscript values for the respective dimensions at some point in time. The lower bound on each range of subscript values will be taken as 0 without loss of generality. Let the state S_i of the array be characterized by d, and u_j, $j = 1, \ldots, d$,. We will assume that d and the u_j's vary in any arbitrary order when they are increasing. Suppose $d = 3$, and the current upper bounds of the respective dimensions are u_1, u_2 and u_3. Let u_1 expand to $u_1 + 1$, followed by an expansion of u_3 to $u_3 + 1$ and suppose that the number of dimensions latter changes from $d = 3$ to $d = 4$. The array is said to go through the following state changes : $S_i = (3; u_1, u_2, u_3)$, $S_{i+1} = (3; u_1+1, u_2, u_3)$, $S_{i+2} = (3; u_1+1, u_2, u_3+1)$, and $S_{i+3} = (4; u_1+1, u_2, u_3+1, u_4)$, where $u_4 = 1$. Variation that decreases the values of d and the $u_j's$ are allowed in so far as the array changes state to the preceding state, i.e,. reverse state transition is allowed in exactly the reverse order of the forward state transitions.

We require a representation of the array A_α in a set of consecutive locations $M[0 : n-1]$, and a mapping function $\Re_\alpha : N^d \rightarrow N$ which maps the elements of A, one-to-one onto the locations $\{M\langle 0\rangle, M\langle 1\rangle, \ldots, M\langle n\rangle\}$ with $A_\alpha\langle 0, \ldots, 0\rangle$ assigned to $M\langle 0\rangle$ called the base location. We imply that the array A_α is dense and rectangular and that n varies in consonance with varying values of d and the u_j's.

1.4 Summary of Main Results.

For simplicity the elements of the array will be considered to be integers, and a unit of storage will be taken to be number of bytes required for an integer. We propose a method , called the $\alpha\beta\gamma$-Index-Array technique, for realizing extendible arrays of varying dimensionality. Let d and n denote the dimensionality and number of elements of an array representation using this scheme. Then the method has the property that

i) The complexity of computing an arbitrary element address is $\Theta(d^2)$.

ii) The storage requirement in the best case is $(n + d^2 n^{1/d} + d)$, units of storage locations when the array is cubical.

iii) The storage requirement in the worst case is $(n + n * d^2/2^{d-1} + d)$ units of storage locations.

If an extendible array has a predefined number of dimensions, d, the $\alpha\beta\gamma$-Index-Array scheme is conservative of storage in comparison with other representation methods that tolerate limited extension. For instance using vector of pointer technique, the corresponding storage requirement for cubical arrays and that array shape for which the $\alpha\beta\gamma$-Index-Array gives the worst case, are respectively $(n + (n - n^{1/d})/(n^{1/d} - 1))$ and $(n(3 - 1/2^{d-1}))$.

The technique for storing extndible array of arbitrary expansions can be carried over for storing two special kinds of extendible arrays that have had immediate application in dynamic data structures structures [5, 6, 7, 8, 11]. These are referred to as "uniform extendible array of linear varying order", and "uniform extendible array of exponential varying order". We shall define the mapping functions for these two types of extendible arrays and show how these functions are modified to admit increasing number of dimensions.

2. Some Definitions and Notations.

We present in this section some formal concepts and a representation independent denotations of an array relevant to our discussions. Let N denote the set of non-negative integers. For some $d \in N - \{0\}$, $u_j \in N$, where j = 1, 2,..., d, let m_j denote the set $\{0, 1, \ldots, u_j\}$ and let $n = \prod_{j=1}^{d} (u_j + 1)$. A bounded rectangular set N_n^d, of d-tuples is defined by the Cartesian product $m_1 \times \ldots \times m_d$. An element in N_n^d is specified by an integer d-tuple $\langle i_1, \ldots, i_d \rangle$, for $0 \leq i_j \leq u_j$, where $j = 1, 2, \ldots, d$. The set of integer d-tuples in N_n^d may be perceived as the the set of all lattice points in the region of d-space defined by $m_1 \times \ldots \times m_d$. The infinite expandability of the array is induced by allowing each set m_j, to expand indefinitely.

Definition 2.1. *Given N, the set of non-negative integers, so that $d, u_j \in N$, for $j = 1, \ldots, d$, and $m_j = \{0, 1, \ldots, u_j\}$, a d-dimensional array $A[0 : u_1, \ldots, 0 : u_d]$, is an association between integer d-tuples and elements of a set E, such that for each d-tuple in the set $m_1 \times \ldots \times m_d$, there corresponds an element of E.*

An element of a d-dimensional array is denoted as $A\langle i_1, \ldots, i_d \rangle$, and consists of that element of E associated with the d-tuple $\langle i_1, \ldots, i_d \rangle$. index. Each individual component i_j of the d-tuple index is referred to as a subscript.

Let $M[0 : n] = \{M\langle 0 \rangle, M\langle 1 \rangle, \ldots, M\langle n \rangle\}$ be a set of consecutive storage locations with

the base location being $M\langle 0\rangle$. An allocation function $\Re : N^d \rightarrow N$, is a mapping that assigns to each element $A\langle i_1, \ldots, \rangle$, a distinct location $M\langle q\rangle$ where q is derived from the array index $\langle i_1, \ldots, i_d\rangle$. The function $\Re$ is said to realize A.

Definition 2.2. *A realization of the* array $A[0 : u_1, \ldots, 0 : u_d]$, *in* $M[0 : n-1]$ *where* $n = \prod_{j=1}^{d} (u_j + 1)$ *is a mapping* $\Re : N^d \rightarrow N$, *of elements of A, one-to-one onto the addresses* $\{0, 1, \ldots, n-1\}$, *such that* $\Re(\langle 0, \ldots, 0\rangle) = 0$.

Examples of realizations of bounded arrays are the row-major and the column-major order functions. These are defined as follows.

(*i*) row-major realization : $\Re_{rm} = s_0 + c_1 i_1 + c_2 i_2 + \ldots + c_d i_d$,

$$\text{where } c_j = \prod_{r=j+1}^{d} (u_j + 1)$$

(*ii*) column-major realization : $\Re_{cm} = s_0 + c_1 i_1 + c_2 i_2 + \ldots + c_d i_d$,

$$\text{where } c_j = \prod_{r=1}^{j-1} (u_j + 1)$$

We follow the convention that an empty product is taken as 1. The row-major order is also referred as the lexicographic order.

Definition 2.3. *The ordering of the d-tuple coordinates for the set of positions in* $m_1 \times \cdots \times m_d$, *where* $m_j = \{0, 1, \ldots, u_j\}$, $j = 1, \ldots, d$, *is said to be lexicographic iff for any two coordinates* $\langle i_1, \ldots, i_d\rangle$ *and* $\langle i'_1, \ldots, i'_d\rangle$ *for which*

$$\langle i_1, \ldots\rangle < \langle i'_1, \ldots, i'_d\rangle$$

there is some k in the range $1 \leq k \leq d$ *such that* $i_j = i'_j$, *for* $1 \leq j < k$ *and* $i_k < i'_k$.

Longitudinal extensions in a d-dimensional array may be percieved as sequence of state transitions in which the size of each dimension is expanded by adjoining a (d-1)-dimensional block. For all purposes the allocation of elements in any adjoined block will be taken to be in column-major order.

Definition 2.4. *A d-dimensional extendible* array *is a d-dimensional* array $A_\alpha[0 : u_1, \ldots, 0 : u_d]$ *in which the upper bounds of the subscripts of the various dimensions may increase in any arbitrary order.*

Definition 2.5. *An extendible array in which the dimensionality d, is varying and takes increasing values of positive integers, is referred to as as an extendible array of varying dimensions.*

An extendible array of varying dimensions is denoted by $A^*_\alpha[0:u_1,\ldots,0:u_d,\ldots]$ where the asterisk ('*'), reflects the fact that the dimensionality is varying and the subscript α, reflects the arbitrary nature of the expansion process. We imply by this that the choice of which dimension to expand next is independent of all other dimensions. In practice, however, some specified criteria will normally dictate the choice of the dimensions to be extended.

Definition 2.6. *A realization of an extendible array of varying dimensions* $A^*_\alpha[0:u_1,\ldots,0:u_d,\ldots]$, *is a function* $\Re^*_\alpha : N^d \rightarrow N$ *that maps elements of* A^*_α *one-to-one onto consecutive storage locations* $M[0:n-1]$ *such that* $\Re^*_\alpha(\langle 0,\ldots,0\rangle) = 0$, *where* $n = \prod_{j=1}^{d}(u_j+1)$, *varies in consonance with changing values of d and* u_j*'s.*

For the case where the dimensionality is constant, we shall drop the asterisk in the notation of the mapping function above. The reader may examine examples of extendible arrays of constant and variable dimensionalities shown in the Figures 3.1a and 3.2a. We present two special cases of arrays having structural regularity in their growth pattern. These are :-

a) uniform extendible array of linear varying order, and

b) uniform extendible array of exponential varying order.

They are defined formally as follows.

Definition 2.7. *Given that* $U_j \in N$, $j = 1,\ldots,d$, *where the* U_j*'s define the limiting values on the upper bounds of the respective dimensions. Let* u_j, *for* $j = 1,\ldots d$, *denote the respective instantaneous upper bounds, such that* $0 \leq u_j \leq U_j$. *An array* $A_\lambda[0:u_1,\ldots,0:u_d]$ *is said to be a uniform extendible array of linear varying order (UXAL), iff for all j, for which* $u_j < U_j$, u_j *is incremented by a constant factor l, at each expansion step.*

The storage mapping function for a UXAL will be denoted by $\Re_\lambda$. The Figure 4.1 depicts the schematic storage layout for a 2-dimensional UXAL when the additive constant

$l = 1$. We shall assume that $l = 1$ in all subsequent discussions unless stated otherwise. The Zig-zag technique of Hansson and Tärnlund [2], and the cubical shell method of Rosenberg [12, 13] are subsumed by the method for UXAL.

Given a UXAL one can envision the case where such an array is allowed to expand laterally by increasing the number of dimensions. When a UXAL is allowed to vary its dimensionality, the corresponding mapping function is denoted by $\Re^*_\lambda$.

Definition 2.8. *Given $H_j \in N$, for $j = 1, \ldots d$, where H_j denotes a predefined maximum exponent of some constant $e > 1$ for dimension j, let $h_j \in N$, denote the instantaneous value of the exponents of e. An array $A_e[0 : u_1, \ldots, 0 : u_d]$, where $u_j = e^{h_j} - 1$, is said to be a uniform extendible array of exponential varying order iff for all j, for which $h_j < H_j$, $(u_j + 1)$ is incremented by a factor e at each expansion step.*

The value of e is referred to as the expansion factor. We shall always assume $e = 2$ in all subsequent discussions of a UXAE, except when we refer to a general case. The Figure 4.3 illustrates the storage layout of a 2-dimensional uniform extendible array of exponential varying for $e = 2$.

Consider the growth of a d-dimensional UXAL. At each expansion step, blocks of subarrays of size equal to the previously allocated array are adjoined along the expandable dimensions in increasing order of the dimension labelling. Consider the j^{th} dimension say, with the subscript range $[0, u_j]$, where $u_j = 2^{h_j} - 1$. When the array expands, by adjoining a subarray along the j^{th} dimension, the upper bound of the subscript range goes from $2^{h_j} - 1$ to $2^{h_j+1} - 1$. The interval of subscript values admitted is $[2^{h_j}, 2^{h_j+1} - 1]$. All other range of subscript values of the dimensions except j remain constant. The block of array elements adjoined in this fashion will be referred to as "the subarray implied by i_j, for all $2^{h_j} \leq i_j \leq 2^{h_j+1} - 1$". In the case of a uniform extendible array of linear varying order with $l = 1$, each subarray block is implied by one and only one subscript value.

The qualities of a realization of each of the three classes of extendible arrays – arrays of arbitrary expansion, arrays of linear expansions and arrays of exponential expansion– will be evaluated using primarily the following three criteria.

1. The complexity of accessing an arbitrary element. For an array of dimensionality d, size n, and a mapping function $\Re$, this measure will be denoted by $\tau(d, n; \Re)$.

2. The worst case storage utilization. This will be denoted by $\sigma(d, n; \Re)$

3. The best case storage requirement where necessary. This will be denoted by $\tilde{\sigma}\ (d, n; \Re)$.

We shall employ the order notations (O- , Θ- and Ω- notations), in the sense defined in the literature.

3. Extendible Arrays of Arbitrary Expansions.

3.1 Fixed Dimensional Arrays.

To appreciate the storage allocation technique for arrays of varying dimensionality that can be arbitrarily extended, let us first briefly outline the Index-Array scheme for extendible arrays where the number of dimensions is constant. The method is fully discussed in [10].

Let $A_\alpha[0 : u_1, \ldots, 0 : u_d]$ be an extendible array. The Index-Array scheme stores A_α, called the principal array in a region of storage $M[0 : n-1]$, and maintains another 3-dimensional array $I_\alpha[0 : U_x, 1 : d, 1 : d]$, where $U_x = max\ (u_1, u_2, \ldots, u_d)$, called the "index array". The index array I_α, is extendible only in the first dimension and as such may be realized, using conventional column-major allocation function, in some region of storage independent of M. The use of the function of the index array is to retain relevant information at each expansion step. This information is used in the computation of the address of an element $A\langle i_1, \ldots, i_d\rangle$, in M.

Let the elements of the array A_α be allocated in locations $M\langle 0\rangle$, $M\langle 1\rangle$, ..., $M\langle n-1\rangle$, and suppose the array is extended by increasing u_j to $u_j + 1$. The array is now denoted as $A_\alpha[0 : u_1, \ldots, 0 : u_j + 1, \ldots, 0 : u_d]$. The effect is that a block of subarray element must be allocated in locations $M\langle n\rangle$, $M\langle n+1\rangle$ through $M\langle n'-1\rangle$ where $n' = (u_j+2) * \prod_{\substack{r=1 \\ r\neq j}}^{d} (u_r+1)$.

Since ordering of the elements in any adjoined block is column-major order, the address q of the location $M\langle q\rangle$ to which an element $A\langle i_1, \ldots, i_d\rangle$ is assigned is given by

$$q = n + \sum_{\substack{k=1 \\ k\neq j}}^{d} c_k * i_k, \quad \text{where} \quad c_k = \prod_{\substack{r=1 \\ r\neq j}}^{k-1} (u_r + 1); \tag{3.1}$$

Let n be the block header address of the adjoined subarray. This is the address of the element $A\langle 0, \ldots, u_j + 1, \ldots, 0\rangle$. The values c_k for $k = 1, \ldots, j-1, j+1, \ldots, d$, in the equation (3.1), are the coefficients with which the respective subscripts must be multiplied to address an element in this block. As a consequence of equation (3.1), we need to retain

the values of the block header address and the required coefficients, whenever the array is extended. These are retained in the index-array I_α.

Let $I_\alpha\langle j_1, j_2, x\rangle$ denote an element of the index array $I_\alpha[1:d, 1:d, 0:U_x]$. A mapping function for realizing I_α, denoted by $\Phi_\alpha : N^3 \rightarrow N$, is defined as

$$\Phi_\alpha(j_1, j_2, x) = (j_1 - 1) + (j_2 - 1) * d + (x - 1) * d^2; \tag{3.2}$$

During the expansion of the array $A_\alpha[0:u_1, \ldots, 0:u_j, \ldots, 0:u_d]$ to $A_\alpha[0:u_1, \ldots, 0: u_j + 1, \ldots, 0 : u_d]$, the starting block header address and the relevant coefficients are entered in the index array as follows. If $(u_j + 1) > U_x$, we set $U_x \leftarrow (u_j + 1)$ and extend the storage locations of I_α by d^2 locations. The block header address is entered into $I_\alpha\langle j, j, u_j + 1\rangle$, and the coefficient $c_k = \prod_{\substack{r=1 \\ r \neq j}}^{k-1}(u_r + 1)$, is entered in $I_\alpha\langle j, k, u_j + 1\rangle$ for $k = 1, \ldots, j-1, j+1, \ldots, d$.

To determine the address of an element $A_\alpha\langle i_1, \ldots, i_d\rangle$, the block in which the element belongs must first be identified. This is straight forward since an element will belong to the lattest of the adjoined blocks implied by the subscript values $i_1, i_2, \ldots, i_d$, i.e., the implied block with the maximum block header address. By comparing the index array entries $I_\alpha\langle j, j, i_j\rangle$, for $j = 1, \ldots, d$, that value of j for which the block header address is maximum is easily determined. The rest of the address computation follows immediately from equation (3.1). Let the function to compute the address of an elements be π_α. Then this is defined algorithmically as

Function $\pi_\alpha(\langle i_1, \ldots, i_d\rangle)$
begin
Determine j such that $I_\alpha\langle j, j, i_j\rangle = max\ (I_\alpha\langle 1, 1, u_1\rangle, \ldots, I_\alpha\langle d, d, i_d\rangle)$;

$$\pi_\alpha \leftarrow I_\alpha\langle j, j, i_j\rangle + \sum_{\substack{k=1 \\ k \neq j}}^{d} (I_\alpha\langle j, k, i_j\rangle * i_k)\ ;$$

end.

To illustrate the addressing method of the Index-Array scheme, consider the Figures 3.1a, and 3.1b which show a 2-dimensional principal array, and the corresponding index array. The Figure 3.1a depicts the storage layout for the following sequence of state transitions : (2; 0, 0), (2; 1, 0), (2; 1, 1), (2: 1, 2), (2; 2, 2), (2; 3, 2), (2; 3, 3). At the time of the last expansion, the upper bound of the dimension i_2 is extended from 2 to

3. The elements allocated are $A_\alpha\langle 0,3\rangle$, $A_\alpha\langle 1,3\rangle$, ..., $A_\alpha\langle 3,3\rangle$. These have the respective addresses 12, 13, ..., 15. The starting block address, which in this case is 12, is stored in $I_\alpha\langle 2,2,3\rangle$ and the coefficient of the subscript i_1, which in this case equals 1, is stored in $I_\alpha\langle 2,1,3\rangle$.

To compute the address of an element $A_\alpha\langle 3,3\rangle$ say, using the function π_α, we first determine which of the entries, $I_\alpha\langle 1,1,3\rangle$ and $I_\alpha\langle 2,2,3\rangle$ is larger. Since $I_\alpha\langle 2,2,3\rangle = 12$ is the larger, we set $j \leftarrow 2$. The final address is given by $\pi_\alpha = I_\alpha\langle 2,2,3\rangle + I_\alpha\langle 2,1,3\rangle * 3 = 12 + 3 = 15$.

Proposition 3.1. *An extendible array of fixed number of dimensions and arbitrary expansion is realisable by the Index-Array scheme, in which the mapping function* $\Re_\alpha = (\pi_\alpha, \Phi_\alpha)$ *is defined by two functions* Φ_α *and* π_α, *where* Φ_α *addresses elements of the index array and* π_α *addresses elements of the principal array.*

The qualities of the Index-Array scheme are summarised by the Theorems 3.2 and 3.3.

Theorem 3.1. *The complexity of computing the address of an arbitrary element in the Index-Array scheme is* $\tau(d,n;\Re_\alpha) = \Theta(d)$.

Proof.

The complexity of the $\Re_\alpha$ computation will be measured by the number of elementary operations such as addition, multiplication, comparisons etc. To access an element in the index array, the evaluation of π_α takes a constant time. Computing $\Re_\alpha$ cost $d-1$ comparisons, d additions and $d-1$ multiplications, hence $\tau(d,n;\Re_\alpha) = \Theta(d)$. ∎

Theorem 3.2. *The Index-Array scheme spreads a d-dimensional extendible array of n elements in* $\sigma(d,n;\Re_\alpha) = O(n\,d^2)$ *locations in the worst case and in* $\breve{\sigma}(d,n:\Re_\alpha) = \Omega(n)$ *locations in the best case.*

Proof.

Consider an array which is predefined as d-dimensional. If circumstances dictate that expansions occur along one dimension always, then the size of the longest dimension is n when the array has n elements. The size of the index array then is $n * d^2$. This is the maximum size that the index array can get for an n-element array. The worst case storage

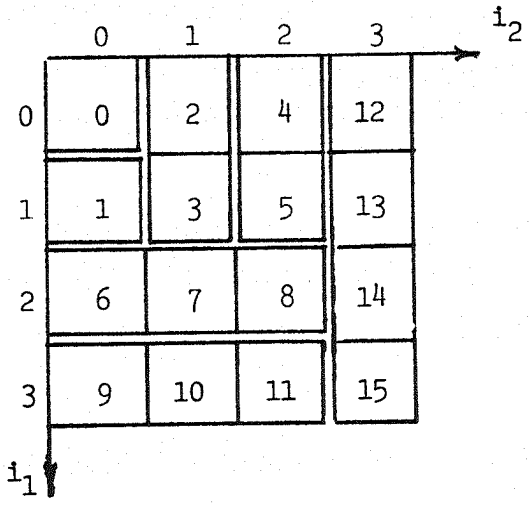

Figure 3.1a : A schematic storage layout of a 2-dimensional extendible array $A_\alpha[0:3, 0:3]$.

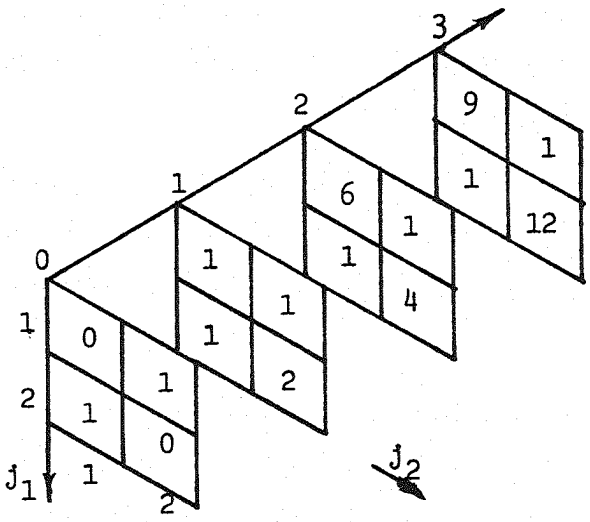

Figure 3.1b : The corresponding index-array of Figure 3.1a showing the entries .

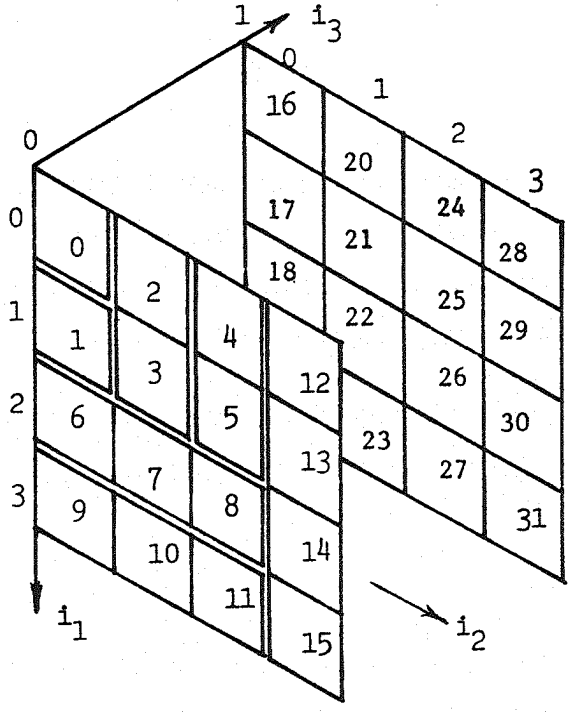

Figure 3.2a : Storage Layout of the array of Figure 3.1a with the number of dimensions extended to 3.

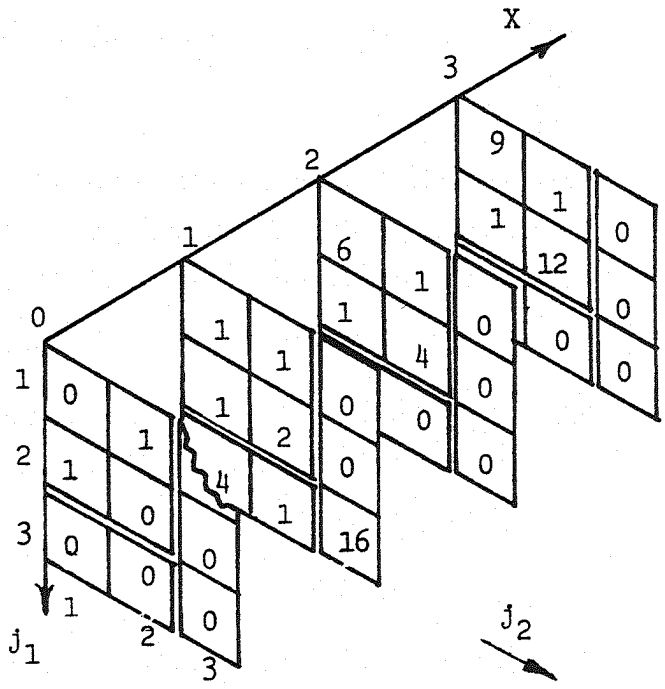

Figure 3.2b : The $\alpha\beta\gamma$-index-array for Figure 3.2a, showing the entries.

utilized then is

$$\sigma(d,n;\Re_\alpha) = n + n * d^2,$$
$$= O(n\,d^2).$$

The minimum size of the index array occurs when the principal array is hypercubical, i.e., each size is $n^{1/d}$. The index array in this case is of size $n^{1/d} * d^2$. The best case storage requirement is

$$\tilde{\sigma}(d,n;\Re_\alpha) = n + n^{1/d} * d^2,$$
$$= \Omega(n). \blacksquare$$

3.2 Arrays of Variable Dimensions.

Using the concept of the Index-Array scheme of the preceding section, the case where the principal array can expand in the number of dimensions is easily appreciated once we understand how the index array I_α can be made extendible in all X, j_1 and j_2 dimensions.

Consider the 2-dimensional extendible array of Figure 3.1a and its associated index array of Figure 3.1b. Suppose the principal array, which we now denote as $A^*_\alpha[0:3,0:3]$, is expanded by increasing the number of dimensions to 3. The new array, shown in Figure 3.2a, is now denoted as $A^*_\alpha[0:3,0:3,0:1]$. This expansion is treated as having occured in two steps. First is the expression of intent to treat the array as a 3-dimensional array, followed by an extension of the subscript range of the third dimension. The effect is to add a 2-dimensional subarray of 16 elements on the third dimension as shown in Figure 3.2a. The added elements in column-major order are $A^*_\alpha\langle 0,0,1\rangle$, $A^*_\alpha\langle 1,0,1\rangle$, $A^*_\alpha\langle 2,0,1\rangle$, ..., $A^*_\alpha\langle 3,3,1\rangle$, and these occupy the addresses 16 through 31.

To store the header address and the required coefficients for addressing elements in the adjoined block, the original index array, denoted now as $I^*_\alpha[1:2,1:2,0:3]$ must be expanded to $I^*_\alpha[1:3,1:3,0:3]$ as shown in Figure 3.2b. The block header address of 16 and the coefficients 4 and 1, can now be stored in the locations $I^*_\alpha\langle 3,3,1\rangle$, $I^*_\alpha\langle 3,1,1\rangle$, and $I^*_\alpha\langle 3,2,1\rangle$ respectively. The added complexity to the original Index-Array scheme, is that the index array must be extendible. We focus attention then on the realization of the index array.

Our strategy for storing extendible arrays of varying dimensions is referred to as the $\alpha\beta\gamma$-Index-Array technique. The array I^*_α of Figure 3.2b is referred to as an $\alpha\beta\gamma$-index-array. Let Φ^*_α denote the mapping function for this array. Consider the the 3-dimensional array of Figure 3.3. This array corresponds to the $\alpha\beta\gamma$-index-array of some unspecified

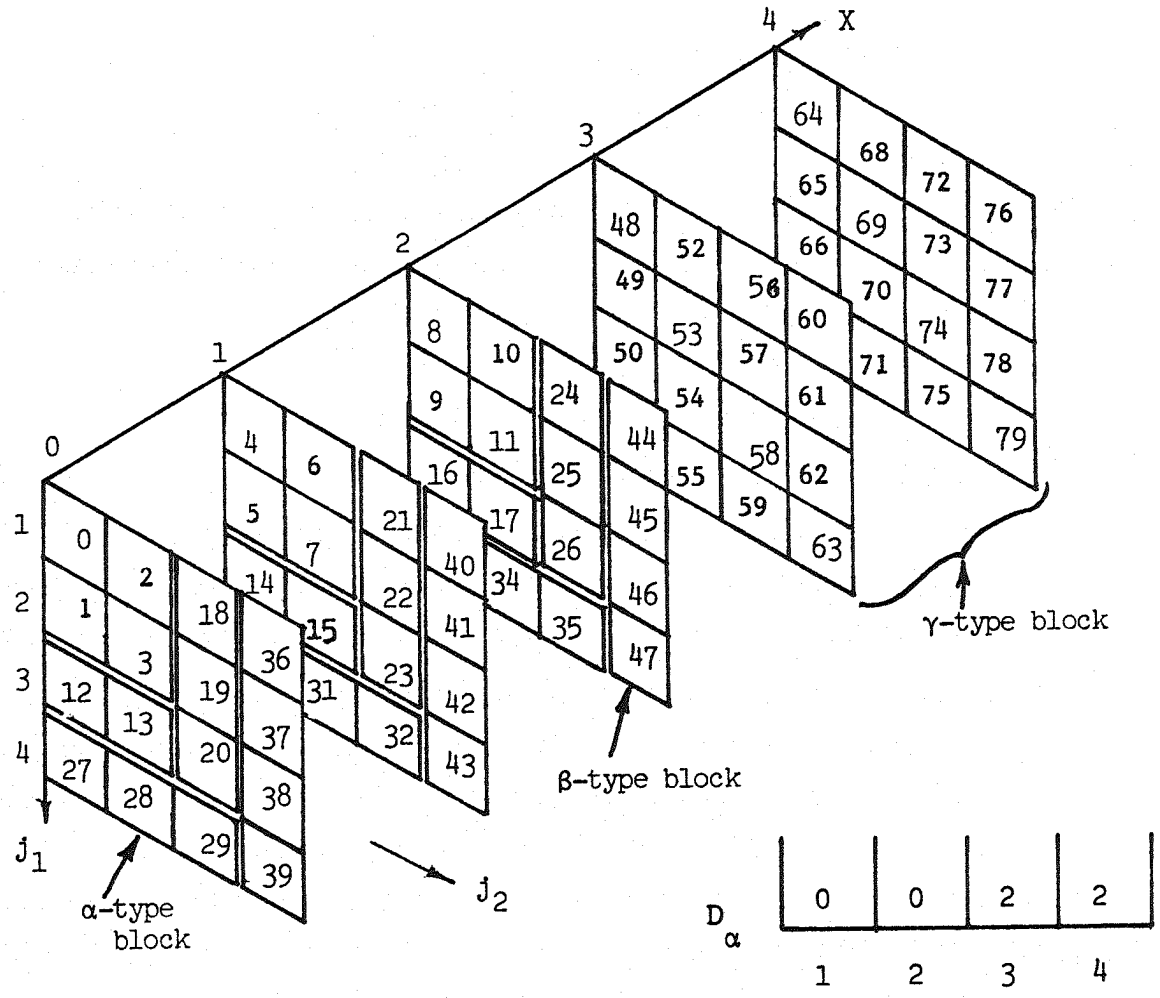

Figure 3.3 : The storage layout of an $a\alpha\beta\gamma$--index-array $I^*_\alpha[1:4,1:4,0:4]$ showing the the addresses of each element.

extendible array of varying dimensions. The three directions of possible expansions are the j_1, j_2 and X dimensions as shown in the Figure 3.3. A characteristic of the $\alpha\beta\gamma$-index-array is that whenever an increase in the number of dimensions of the principal array is expressed, the sizes of the j_1 and j_2 dimensions increase such that both remain equal.

To address elements in the index array correctly, we maintain a third array $D_\alpha[1:d]$, which is 1-dimensional and is of size that is equal to the number of dimensions of the principalarray at all times. The entries in the D_α array are the values of U_x of the index

array at the instants the dimensionality of the principal array is increased.

Suppose the dimensionality of the principal array A^*_α is increased from d to (d + 1). The j_1 and j_2 dimensions of the $\alpha\beta\gamma$ index array are immediately increased from d to (d + 1) by adjoining a subarray of $d * U_x$ elements along the j_1 dimension followed by the adjunction of a subarray of $(d+1) * U_x$ elements on the j_2 dimension. The subarray blocks adjoined on the j_1 and j_2 dimensions are referred to as the " α-type" and "β-type" blocks respectively. The initial values of all entries of the adjoined block of elements is zero. Between any two consecutive instances of increasing the dimensionality of the principal array, extensions of the range of subscript values may occur. Such extensions may increase the size of the U_x dimension of the index array. The block of index array elements added to extend the size U_x, is said to constitute a "γ-type" block. The Figure 3.3 distinguishes the three types of blocks of the index array. For the case where the extendible array admits variation in the number of dimensions, the index array then, is comprised of $\alpha-$, $\beta-$ and γ-type blocks, hence the name $\alpha\beta\gamma$ index array.

To determine the address of an index array element $I^*_\alpha\langle j_1, j_2, x\rangle$, we need to know the dimensionality at the time that element was admitted and what block type it belongs to. Define a function $\Psi_\alpha(x)$ that associates a value δ, $1 \leq \delta \leq d$, for any integer x, where $0 \leq x \leq U_x$, as follows.

$$\Psi_\alpha(x) = \text{maximum} d \text{ such that} D_\alpha(d) \leq x.$$

The function Ψ_α can be perceived as an inverse of a subscript function. The mapping function for addressing elements of the $\alpha\beta\gamma$-index-array is defined as follows.

Function $\Phi^*_\alpha(j_1, j_2, x)$;
begin
  if $j_1 > j_2$ **then**
    if $D_\alpha\langle j_1\rangle \geq x$ **then**
      begin $\delta \leftarrow j_1$; $h \leftarrow D_\alpha\langle j_1\rangle$; *block_type* $\leftarrow$ *α-type*; **end**;
    else
      begin $\delta \leftarrow \Psi_\alpha(x)$; *block_type* $\leftarrow$ *γ-type*; **end**
  else
    if $D_\alpha\langle j_2\rangle \geq x$ **then**
      begin $\delta \leftarrow j_2$; $h \leftarrow D_\alpha\langle j_2\rangle$; *block_type* $\leftarrow$ *β-type* ; **end**
  else

begin $\delta \leftarrow \Psi_\alpha(x)$; *block_type* $\leftarrow$ γ-*type*; **end**

case block_type of

γ-*type* : $q \leftarrow (j_1 - 1) + (j_2 - 1) * \delta + x * \delta^2$;

α-*type* : $q \leftarrow (j_2 + 1) + x * (\delta - 1) + (h + 1) * (\delta - 1)^2$;

β-*type* : $q \leftarrow (j_1 + 1) + x * \delta + (h + 1) * \delta * (\delta - 1)$;

$\Phi^*_\alpha \leftarrow q$;

end ;

The computation carried out by the Φ^*_α function is illustrated with the following example. Suppose in Figure 3.3 we wish to access the location of the element $I^*_\alpha\langle 3,3,3\rangle$. We have then that $x = 3$, $j_1 = 3$, and $j_2 = 3$. Since $j_1 \leq j_2$ and $D_\alpha\langle j_2\rangle < x$, we set $\delta \leftarrow \Psi_\alpha(x) = \Psi_\alpha(3) = 4$, and *block_type* $\leftarrow$ γ-type. The evaluation of $\Phi^*_\alpha(3,3,3)$ then is given by

$$\Phi^*_\alpha(3,3,3) = (j_1 - 1) + (j_2 + 1) * \delta + x * \delta^2$$
$$= 2 + 2 * 4 + 3 * 4^2 = 58.$$

To compute the address of an arbitrary element in the $\alpha\beta\gamma$-Index-Array scheme, we apply the π_α function as in the Index-Array scheme, except that now elements in the $\alpha\beta\gamma$ index array are addressed using the function Φ^*_α.

Proposition 3.2. *An extendible array of varying dimensions is realizable, using the $\alpha\beta\gamma$-Index-Array scheme, by $\Re^*_\alpha = (\pi_\alpha, \Phi^*_\alpha, \Psi_\alpha)$, where π_α is the mapping function for the elements of the principal array, Φ^*_α is the mapping function for the elements in the $\alpha\beta\gamma$-index-array and Ψ_α is an inverse subscript function for the array D_α.*

Theorem 3.3. *The complexity of computing the address of an arbitrary element in an extendible array of varying dimensions having n elements in d dimensions is $\tau(d, n; \Re^*_\alpha) = \Theta(d^2)$.*

Proof.

The computation of Φ^*_α is $\Theta(d)$, since every statement involves a constant number of elementary operations except for the evaluation of Ψ_α which takes at most $d-1$ comparisons. In computing π_α, the funtion Φ^*_α is computed d times to determine the block header address and a further d-1 times in the computation of sum of product terms. The total time to evaluate $\Re^*_\alpha$ then is a constant sum of $d * \Theta(d)$ terms which is $\Theta(d^2)$. ∎

Theorem 3.4. *The $\alpha\beta\gamma$-Index-Array scheme realizes a d-dimensional extendible array*

of variable dimensions having n elements using $\sigma(d, n; \Re^*_\alpha) = \Theta(n)$ *storage locations.*

Proof.

The n elements of the principal array A^*_α are allocated in exactly n storage locations. Whenever the number of dimensions increases, the subscript range of the admitted dimensions becomes 2. Hence the size of each dimension is at least 2 except for one which is at most $n/2^{d-1}$. The maximum storage required for the $\alpha\beta\gamma$ index array is therefore $n * d^2/2^{d-1}$. The total storage requirement is at most $n + n * d^2/2^{d-1} + d$, which is $O(n)$.

The size of the $\alpha\beta\gamma$-index-array is minimum if and only if the principal array is hypercubical. This implies that the size of each index range is $n^{1/d}$. Consequently the minimum storage requirement for the index array is $n^{1/d} * d^2$ and the minimum total storage required in the realization is $n + n^{1/d} * d^2 + d$, which is $O(n)$. The two results of the maximum and minimum storage requirements imply that $\sigma(d, n; \Re^*_\alpha) = \Theta(n)$. ■

Corollary 3.1. *The* $\alpha\beta\gamma$*-Index-Array scheme, favours infinite family of shapes such that an array of n elements is spread over* $\sigma(d, n; \Re^*_\alpha) \leq 3.25\, n$ *locations for* $n >> d$.

Proof.

The follows from the proof of Theorem 3.4. We have that the maximum storage required for any rectangular array of any shape is $\sigma(d, n; \Re^*_\alpha) = n + n * d^2/2^{d-1} + d$. This expression is maximum when $d = 3$, for $n >> d$. Hence we have that $\sigma(d, n; \Re^*_\alpha) \leq 3.25\, n$. ■

Note that the Index-Array scheme may be considered as a special case of the $\alpha\beta\gamma$-Index-Array scheme in which the dimensionality is bounded. Although this gives an improved worst case storage utilization, an element access cost is now $\Theta(d^2)$ instead of $\Theta(d)$. Other qualities of the scheme are summarized below.

i) Ease of traversal along paths.

Given that the subscript i_j of the j^{th} dimension steps through the values 0, 1, 2, ..., u_j, while others are held constant, a traversal is said to be easy if the elements can be found at addresses that increase in some arithmetic progression. The $\alpha\beta\gamma$-Index-Array method allows such additive traversal only within an adjoined subarray block. In general, the addresses of the elements along such a path must be computed individually.

ii) Ease of longitudinal extension.

Extending the subscript range of any dimensions is easy since the elements of the subarray adjoined are allocated in contiguous locations following the storage area of the previously allocated array with no reorganization. The same easy extensions are made to the $\alpha\beta\gamma$-index-array and the array D_α.

iii) Ease of lateral extension.

The $\alpha\beta\gamma$-Index-Array scheme is the only method known yet for managing storage for extendible arrays, such that the element access cost is independent of the number of elements in the array and further allows the number of dimensions to be varied.

We compare the $\alpha\beta\gamma$-Index-Array method with the row-major order and linked allocation using vector of pointers, of representing arrays, since these are techniques in which an element access cost is independent of the size of the array and are reasonably conservative in storage utilization. The Table 3.1 summarizes the result of the comparison of the three approaches of representing arrays.

4. Arrays With Structural Regularity.

4.1. Uniform Arrays With Linear Expansion.

In Section 2. a uniform extendible array of linear varying order (UXAL), was defined as an example of an array that expands to maintain some structural regularity. Consider first the case where the dimensionality is constant. Let $A_\lambda[0:u_1,\ldots,0:u_d]$ denote a UXAL, where for some predefined maximum values, U_j, $j = 1,\ldots,d$, we have $0 \leq u_j \leq U_j$. Suppose $d = 2$, $U_1 = 4$, $U_2 = \infty$, then the Figure 4.1 shows the schematic storage layout after the array progressively expands from $A_\lambda[0:0,0:0]$ to $A_\lambda[0:4,\ 0:6]$. Note that u_1 has attained its maximum value of 4, and since $U_2 = \infty$, subsequent extensions only occur on the second dimensions.

The realization of a d-dimesnensional UXAL is given by $\Re_\lambda$ which is defined below.

Function $\Re_\lambda(\langle i_1,\ldots,i_d\rangle)$;

{ l is an additive constant of the subscript range.}

begin

L1. Set $t \leftarrow$ largest j such that $\lceil i_j/l\rceil = \max(\lceil i_1/l\rceil,\ldots,\lceil i_d/l\rceil)$;

L2. Set $q \leftarrow \lceil i_t/l\rceil$;

L3. **for** $j \leftarrow 1$ **to** d **do**

Characteristic	Lexicographic Allocat--ion method	Vector of Pointers method	$\alpha\beta\gamma$-Index-Array method
Complexity of element access.	$\Theta(d)$	$\Theta(d)$	$\Theta(d^2)$
Storage required.			
best case $\tilde{\sigma}$	n	$n + (\frac{n-n^{1/d}}{n^{1/d}-1})$	$n + d^2\, n^{1/d} + d$
worst case σ	n	$n(3 - 1/2^{d-1})$	$n(1 + d^2/2^{d-1}) + d$
Ease of traversal along paths	Easy traversal of elements in a hyper-plane orthognal to an axis.	Traversal along a path by successive addition of a const--ant is easy in last dimension.	Traversal of elements in a hyperplane orthogonal to an axis is possible only within a block.
Ease of Longitud--inal extension.	Admits extensions in the first dimension.	Admits extensions in the last dimensions	Allows extensions on any dimension.
Ease of lateral extension.	Not allowed.	Not easily done.	Allows increase in the number of dimensions.

Table 3.1 : A comparison of three methods of array representations.

if $j < t$ **then**

$J_j \leftarrow \min(q * l + 1,\ U_j + 1)$

else

$J_j \leftarrow \min((q-1) * l + 1,\ U_j + 1)$;

L4. **for** $j \leftarrow 1$ **to** d, $j \neq t$ **do**

$$c_j \leftarrow \prod_{\substack{r=j+1 \\ r \neq t}}^{d} J_r;$$

L5. $$\Re_\lambda \leftarrow i_t * \prod_{\substack{j=1 \\ j \neq t}}^{d} J_j + \sum_{\substack{j=1 \\ j \neq t}}^{d} c_j * i_j;$$

end.

Theorem 4.1. *The function $\Re_\lambda$ is a realization of a d-dimensional uniform extendible array of linear varying order such that an array of n elements is spread in $\sigma(d, n; \Re_\lambda) = \Theta(n)$ locations and an arbitrary element is accessed in time $\tau(d, n; \Re_\lambda) = \Theta(d)$.*

Proof.

An n element array is allocated in exactly n locations so that $\sigma(d, n; \Re_\lambda) = \Theta(n)$. To compute the function $\Re_\lambda$, each of the statements L1, L3, L4 and L5, is evaluted in $\Theta(d)$ elementary operations of additions, multiplications and comparisons. Hence the time complexity to compute $\Re_\lambda$ is given by $\tau(d, n; \Re_\lambda) = \Theta(d^2)$. ∎

4.2. Uniform Arrays With Linear Expansion and Variable Dimensions.

Consider a unifrom extendible array of linear varying order in which the number of dimensions is allowed to vary. Let the upper bounds of each dimension be infinitly extendible, and suppose the UXAL is initialized as a d-dimensional array which is denoted by $A^*_\alpha[0 : u_1, \ldots, 0 : u_d]$. At each expansions step, the subscript range each dimension is extended by l. Consider that at each expansion step, (d-1)-dimensional array blocks are adjoined along the dimensions in the order $i_1, i_2, \ldots, i_d$. Let the dimensionality of the array be increased from d to $(d+1)$. The array may first be denoted as $A^*_\lambda[0 : u_1, \ldots, 0 : u_d, 0 : 0]$. At the next expansion, when every subscript range is increased by one, the new dimension added will also be expanded. The condition on the upper bounds of the subscripts is $u_1 = u_2 = \ldots = u_d \neq u_{d+1}$. A uniform extendible array of linear varying order with variable dimensions is referred to in short, as a $UXAL^*$.

Although the the $\alpha\beta\gamma$-Index-Array scheme may be used to realize a $UXAL^*$, the structural regularity in the expansion allows the definition of a mapping function $\Re^*_\lambda$ that addresses elements in a more compact storage location than the $\alpha\beta\gamma$-Index-Array scheme would permit.

The $\Re^*_\lambda$ is a variant of the $\Re_\lambda$ that uses an extendible triangular array $D_\lambda[1 : d, 1 : d]$, to hold information about the sizes of the dimensions, whenever an increase in the number of dimensions is expressed. An element of the array D_λ is denoted by $D_\lambda\langle j_1, j_2\rangle$, $1 \leq j_1 \leq j_2 \leq d$. Consider the instant when a d-dimensional array $A^*_\lambda[0 : u_1, \ldots, 0 : u_d]$ increases its dimensionality to become $A^*_\lambda[0 : u_1, \ldots, 0 : u_d, 0 : 0]$. We extend the array $D_\lambda[1 : d,\ 1 : d]$ to $D_\lambda[1 : (d+1),\ 1 : (d+1)]$ and assign the values $u_1,\ u_2, \ldots, u_d$ to $D_\lambda\langle d+1, 1\rangle$, $D_\lambda\langle d+1, 2\rangle$, ..., $D_\lambda\langle d+1, d\rangle$ respectively.

The Figure 4.2a illustrates the allocation of the elements of a $UXAL^*$ which may now be denoted as as a 3-dimensionsal array $A^*_\lambda[0 : 5,\ 0 : 4,\ 0 : 2]$. The Figure 4.2b shows the content of the corresponding array D_λ. Assuming the array was initially declared as

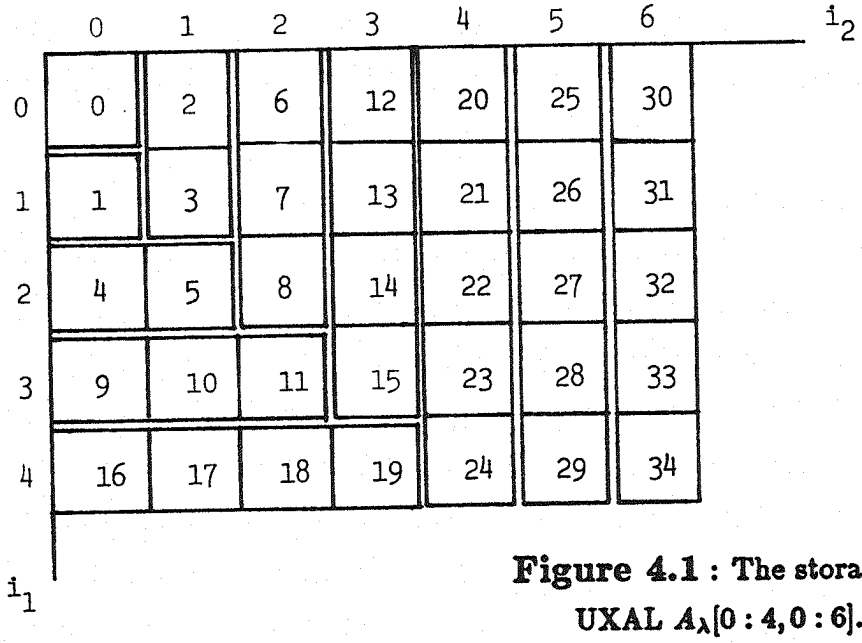

Figure 4.1 : The storage layout of a 2-dimensional UXAL $A_\lambda[0:4, 0:6]$.

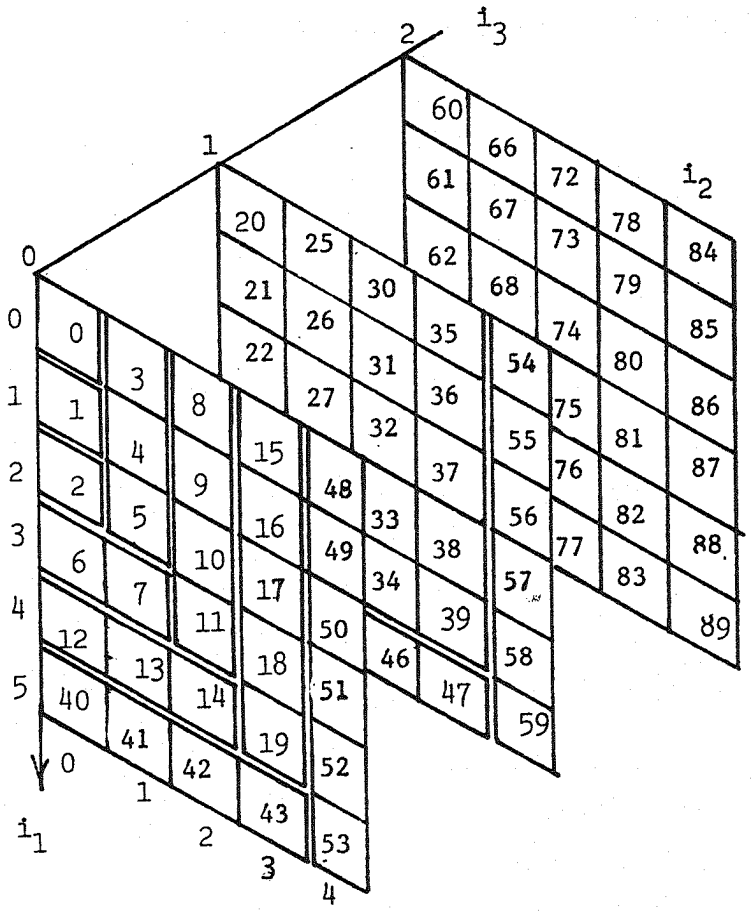

Figure 4.2a : The storage layout of a UXAL of varying dimensions.

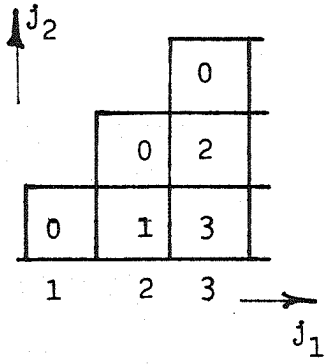

Figure 4.2b : The corresponding D_λ array of Figure 4.2a showing the stored entries.

a one dimensional array $A^*_\lambda[0:0]$, then state transitions of the array shown in Figure 4.2a is described by the sequence (1; 0), (1; 1), (2; 1,0), (2; 2,1), (2; 3,2), (3; 3,2,0), (3; 4,3,1), (3; 5,4,2).

Using the information retained in D_λ, the address of any element $A^*_\lambda\langle i_1,\ldots,i_d\rangle$, when the principal array has dimensionality d, is computed by the mapping function $\Re^*_\lambda$. The idea behind the mapping functions is to determine the starting block header addresses S_1, $S_2,\ldots,S_d$ corresponding to the address of the elements $A^*_\lambda\langle z_1,0,\ldots,0\rangle$, $A^*_\lambda\langle 0,z_2,0,\ldots,0\rangle$ ..., $A^*_\lambda\langle 0,0,\ldots,z_d\rangle$ respectively, where $z_j = (\lceil i_j/l\rceil - 1) * l$, for $j = 1,\ldots,d$. The maximum value S_t gives the block header address of that block to which the element belongs. This is the block implied by i_t. The rest of the computation in $\Re^*_\lambda$ determines the multiplying coefficients c_j, for $j = 1,\ldots,d$, and $j \neq t$, required to address elements in the block implied by i_t. The address is eventually computed as a sum of product terms $\sum_{\substack{j=1\\ j\neq t}}^{d} c_j * i_j$, which is added to the block header address. Let $\Psi_\lambda(x,y)$ denote an inverse subscript function of the array D_λ, which is defined as follows.

Function $\Psi_\lambda(x,y)$;

If $x = 0$ **then** $\Psi_\lambda \leftarrow 1$;

else $\Psi_\lambda \leftarrow$ maximum j_1 such that $D_\lambda\langle j_1,y\rangle < x$;

The function $\Re^*_\lambda$ is defined below .

Function $\Re^*_\lambda(\langle i_1,\ldots,i_d\rangle)$

R1. For $j = 1,2,\ldots,d$ do the following.

Set $\delta_j \leftarrow \Psi_\lambda(i_j,j)$; $k \leftarrow max\ (\lceil i_j - D_\lambda\langle\delta_j,j\rangle overl\rceil - 1,0)$;

and $S_j \leftarrow \prod_{r=1}^{j-1} min\ (D_\lambda\langle\delta_j,r\rangle + (s+) * l + 1, U_r + 1)$

$$* \prod_{r=j}^{\delta_j} (\min(D_\lambda\langle\delta_j,r\rangle + s * l + 1, U_r + 1);$$

R2. Set t $\leftarrow$ highest j such that $S_j = \max(S_1,\ldots,S_d)$;

R3. For $r = 1,2,\ldots,\delta_t$ do the following.

Set $k \leftarrow max\ (\lceil i_r - D_\lambda\langle\delta_t,r\rangle overl\rceil - 1,0)$;

if r ¡ t then set $J_r \leftarrow \min(D_\lambda\langle\delta_t,r\rangle + (k+1) * l + 1, U_r + 1)$;

else $J_r \leftarrow \min(D_\lambda\langle\delta_t,r\rangle + k * l + 1, U_r + 1)$;

R4. for $j = 1, 2, \ldots, \delta_t$, set $c_j \leftarrow \prod_{\substack{r=1 \\ r \neq t}}^{j-1} J_r$;

R5. $\Re^*_\lambda \leftarrow i_t * \prod_{\substack{j=1 \\ j \neq t}}^{\delta_t} J_j + \sum_{\substack{j=1 \\ j \neq t}}^{\delta_t} (c_j * i_j)$;

end;

Theorem 4.2. *The mapping function $\Re^*_\lambda$ realises a uniform extendible array of linear varying order and variable number of dimensions, using $\sigma(d, n : \Re^*_\lambda) = \Theta(n)$ storage locations with $\tau(d, n; \Re^*_\lambda) = \Theta(d^2)$ elements access complexity where d and n are the respective dimesionality and size of the array.*

Proof.

For any n-element array for which the dimensionality is d, the number of storage locations required for the principal array A^*_λ, is n. The nuumber of locations used in representing the array D_λ is $d * (d+1)/2$. The total storage required then is $n + d * (d+1)/2$. Hence $\sigma(d, n; \Re^*_\lambda) = \Theta(n)$.

In the computation of $\Re^*_\lambda$, the statement R1 is evaluated in time $\Theta(d^2)$. All others are each evaluated in time $\Theta(d)$. Hence the total computation, which formed from the sum of a constant number of $\Theta(d^2)$ and $\Theta(d)$ terms, is given by $\tau(d, n; \Re^*_\lambda) = \Theta(d^2)$. ■

4.3 Uniform Arrays With Exponential Expansion.

A second form of extendible arrays with a prescribed pattern of growth is the uniform extendible array of exponential varying order (UXAE), defined in section 2. The primary characteristic of this array is that at each expansion step, the range of subscript value for each dimension expands by a multiplicative factor e. This is in contrast to the UXAL in which the expansion is by an additive constant factor l.

Consider the case where the factor $e = 2$, and the diemnsionality d is constant. Such an array has application in the design of a multidimensional extendible hashing scheme [5, 6, 8]. Let a UXAE be denoted as $A_e[0 : u_1, \ldots, 0 : u_d]$, where $u_j = e^{h_j} - 1$, $0 \leq h_j \leq H_j$ for $j = 1, \ldots, d$. The H_j's are predefined maximum values that the $h'_j s$ can attain in the course of expansions. The Figure 4.3 shows the storage configuration of a 2-dimensional UXAE, with $e = 2$. This is denoted by $A_e[0 : 7, 0 : 7]$. Suppose we have $h_1 = 3$, $H_1 = 3$, $h_2 = 4$, and $H_2 = \infty$. Since $h_1 = H_2$, further expansions can only increase the range of the

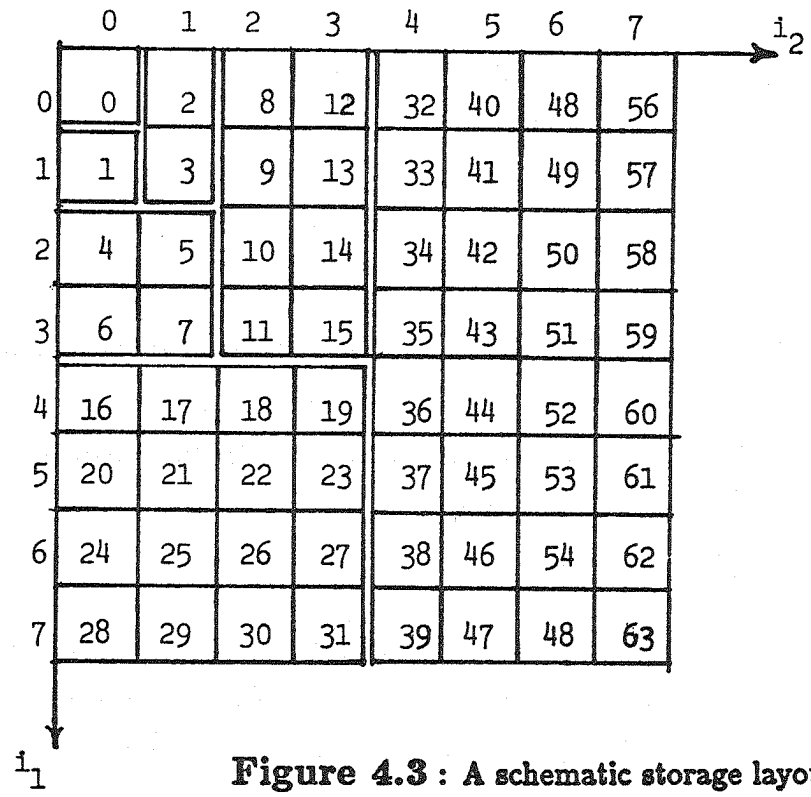

Figure 4.3 : A schematic storage layout of a UXAE $A_e[0:7, 0:7]$ showing the addresses.

subscript i_2.

The realization of a UXAE is given by the mapping function $\Re_e$ defined below.

Function $\Re_e(\langle i_1, \ldots, i_d \rangle)$
begin
E1. Set $t \leftarrow$ highest j such that $\lfloor \log_e i_j \rfloor = \max(\lfloor \log_e \rfloor, \ldots, \lfloor \log_e i_d \rfloor)$;
E2. **if** $i_t = 0$ **then**
set $\Re_e \leftarrow 0$;
else begin
E3. $h_t \leftarrow \lfloor \log_e i_t \rfloor$;
E4. **for** $j \leftarrow 1$ **to** d **do**
if $j < t$ **then**
$J_j \leftarrow \min(e^{h_t+1}, e^{H_j})$;
else
$J_j \leftarrow \min(e^{h_t}, e^{H_j})$;
E5. **for** $j \leftarrow 1$ **to** d **do**

$$c_j \leftarrow \prod_{\substack{r=1 \\ r \neq t}}^{j-1} J_r;$$

E6. $$\mathfrak{R}_e \leftarrow i_t * \prod_{\substack{j=1 \\ j \neq d}}^{d} J_r + \sum_{\substack{j=1 \\ j \neq t}}^{d} c_j * i_j;$$

end;

end;

Theorem 4.3. *The mapping function $\mathfrak{R}_e$, realizes a uniform extendible array of exponential varying order in d dimensions, such that the storage requirement for an n-element array is $\sigma(d, n; \mathfrak{R}_e) = \Theta(n)$, and an arbitrary element is accessed in time $\tau(d, n; \mathfrak{R}_e) = O(d^2 + d \lg X)$, where $\lfloor \log_e X \rfloor = \max(\lfloor \log_e i_1 \rfloor, \ldots, \lfloor \log_e i_d \rfloor)$.*

Proof.

The number of locations used to represent a UXAE array of n elements is exactly n, hence $\sigma(d, n; \mathfrak{R}_e) = \Theta(n)$.

Take the logarithmic operation as an elementary operation. Then the computation of $\mathfrak{R}_e$ is determined by the evaluation times of statement E4, which is $O(d \log_e X)$, where X is one of the subscripts $i_1, \ldots, i_d$ such that $\lfloor \log_e X \rfloor = \max(\lfloor \log_e i_1 \rfloor, \ldots, \lfloor \log_e i_d \rfloor)$. All others are computed in $\Theta(d)$ time. The complexity of $\mathfrak{R}_e$ then is given by $\tau(d, n; \mathfrak{R}_e) = O(d \lg X)$.

■

The function $\mathfrak{R}_e$ can be modified to define a variant $\mathfrak{R}_e^*$ that realizes an extendible array of exponential varying order and of varying dimensions. We leave the reader to define the apropriate function by mimicking the technique for the corresponding $\mathfrak{R}_\lambda^*$, which uses a triangular array D_λ. Equivalently, the function $\mathfrak{R}_e^*$ is easily established with the aid of a triangular array D_e.

5. Conclusion.

Although conventional mapping functions for realizing rectangular arrays of fixed dimensionality allow entensions of the array only along one dimension it is possible, using the basic row-major function, to realize arrays that allow the size of any dimension, as well as the number of dimensions to grow. We have demonstrated how this is achieved using the $\alpha\beta\gamma$-Index-Array scheme in which an array of n elements is represented in at most $3.25\,n$ storage locations and an arbitrary element access cost is $\Theta(d^2)$.

Where some structural regularity in the pattern of growth can be identified, one can achieve a realization of an n element array of d dimensions in $\Theta(n)$ locations with $\Theta(d^2)$ element access cost. Examples of such arrays are illustrated by i) uniform extendible array of linear varying order, and ii) uniform extendible array of exponential varying order.

The results presented in the paper solves the open question of representing an extendible array of n elements in consecutive memory locations such that an array of any shape may be represented in Cn locations. We have not only shown that $C \leq 3.25$, but we have also given the solution to the case when the number of dimensions is allowed to vary. The techniques presented in this paper establish a general framework for realizing extendible arrays.

Acknowledgement.

We thank Nick Briggs for his assistance in the typing of this paper. This research is supported in part by the Natural Sciences and Engineering Research Council of Canada, grant No NSERCC A0317-102B.

References.

[1] De Millo, R. A. Eisenstat, S. C. and Lipton, R. J. Preserving Average Proximity in Arrays. *CACM 21, 3 (March. 1978), 228–231.*

[2] Hansson, Ake, and Tärnlund, Sten-Ake. Ziz-zag Procedures for Memory Allocation and Retrieval of Dense Equilateral Arrays of Dynaically Varying Order. *Bit 16 (1976), 269–274.*

[3] Knott, G. D. Procedures for Managing Extendible Array Files. *Software-Practice and Experience 11 (1981), 63–84 .*

[4] Knuth, D. E. The Art of Computer Programming. Vol. 1 : Fundamental Algorithms. *Addison-Wesley, Reading, Massachusetts, 1968.*

[5] 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 (December 1981).*

[6] Otoo, Ekow J. Low Level Structures in the Implementation of the Relational Algebra. *Ph.D Dissertation (August 1983), McGill University, Montreal, Canada.*

[7] Otoo, Ekow J. A Method for Associative Searching Using Index Tries. *School of Computer Science, Technical Report, Carleton University, Ottawa.*

[8] Otoo, Ekow J. A Mapping Function for the Directory of a Multidimensional Extendible Hashing. *Proc. of the 10th International Conference on Very Large Data Bases (August 1984), Singapore.*

[9] Otoo, E. J. and Merrett, T. H. Dynamic Multipaging: A Multidimensional Structure for Fast Associative Searching. *School of Computer Science, Technical Report No SCS-TR-56, Carleton University, Ottawa, Canada. Submitted* .

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

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

[12] Rosenberg, A. L. Allocating Storage for Extendible Arrays. *J. ACM 21, 4 (October 1974), 652–670.*

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

[14] Rosenberg, A. L. and Stockmeyer, L. J. Hashing Schemes for Extendible Arrays. *J. ACM 24, 2 (April 1977), 199–221.*

[15] Standish, T. A. Data Structure Techniques. *Addison-Wesley, Reading, Massachusetts, 1980.*

[16] Stockmeyer, L. J. Extendible Array Realization with Additive Traversal. *Report RC-4578, IBM Thomas J. Watson Research Centre, Yorktown Heights, N. Y., 1973.*