Thesis No: CSER-M-12-01

AN EFFICIENT IMPLEMENTATION SCHEME FOR MULTIDIMENSIONAL INDEX ARRAY OPERATIONS AND ITS EVALUATION

By

Sheikh Mohammad Masudul Ahsan

Department of Computer Science and Engineering
Khulna University of Engineering & Technology
Khulna 9203, Bangladesh
January, 2012

An Efficient Implementation Scheme for Multidimensional Index Array Operations and Its Evaluation

By

Sheikh Mohammad Masudul Ahsan
Roll No: 0907501

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science & Engineering

Department of Computer Science and Engineering
Khulna University of Engineering & Technology
Khulna 9203, Bangladesh
January, 2012

Declaration

This is to certify that the thesis work entitled “An Efficient Implementation Scheme for Multidimensional Index Array Operations and Its Evaluation” has been carried out by Sheikh Mohammad Masudul Ahsan in the Department of Computer Science and Engineering, Khulna University of Engineering & Technology, Khulna, Bangladesh. The above thesis work or any part of this work has not been submitted anywhere for the award of any degree or diploma.

Signature of Supervisor

Signature of Candidate

Approval

This is to certify that the thesis work submitted by Sheikh Mohammad Masudul Ahsan entitled "An Efficient Implementation Scheme for Multidimensional Index Array Operations and Its Evaluation" has been approved by the board of examiners for the partial fulfillment of the requirements for the degree of Master of Science in Computer Science & Engineering in the Department of Computer Science and Engineering, Khulna University of Engineering & Technology, Khulna, Bangladesh in January, 2012.

BOARD OF EXAMINERS

1. _____
   Dr. K. M. Azharul Hasan — Chairman (Supervisor)
   Professor, Dept. of CSE
   Khulna University of Engineering & Technology, Khulna

2. _____
   Head of the Department — Member
   Department of Computer Science and Engineering
   Khulna University of Engineering & Technology, Khulna

3. _____
   Dr. Muhammad Sheikh Sadi — Member
   Associate Professor, Dept. of CSE
   Khulna University of Engineering & Technology, Khulna

4. _____
   Dr. Kazi Md. Rokibul Alam — Member
   Associate Professor, Dept. of CSE
   Khulna University of Engineering & Technology, Khulna

5. _____
   Dr. Rameswar Debnath — Member (External)
   Professor, Dept. of CSE
   Khulna University, Khulna

Acknowledgment

All the praise to the almighty Allah, whose blessing and mercy succeeded me to complete this thesis work fairly. I gratefully acknowledge the valuable suggestions, advice and sincere co-operation of Dr. K. M. Azharul Hasan, Professor, Department of Computer Science and Engineering, Khulna University of Engineering & Technology, under whose supervision this work was carried out. His open-minded way of thinking, encouragement and trust makes me feel confident to go through different research ideas. From him, I have learned that scientific endeavor means much more than conceiving nice algorithms and to have a much broader view at problems from different perspectives. I would like to convey my heartily ovation to all the faculty members, officials and staffs of the Department of Computer Science and Engineering as they have always extended their co-operation to complete this work. I am extremely indebted to the members of my examination committee for their constructive comments on this manuscript. Last but not least, I wish to thank my friends and my family for their constant support.

Author

Abstract

Multidimensional arrays are greatly used for handling large amount of data in scientific or engineering, and Database applications. Most of the on hand data structures are static in nature. We describe a novel implementation idea of multidimensional array for handling such large scale datasets. The scheme implements a dynamic multidimensional extendible array employing a set of two dimensional extendible arrays. The Traditional Multidimensional Array (TMA) or Extended Karnaugh Map Represented (EKMR) array is an efficient structure in terms of accessing the element of the array by straight computation of the addressing function, but they are not extendible during run time. But real world data grows in incremental fashion. So, there is strong demand of data structure that is dynamically extendible during run time. Three are some extendible array models, most of which uses a concept of extension subarray. For *n*-dimensional array the subarrays are *n*-1 dimensional. But, if the length of dimension and/or number of dimension of a multidimensional array is large then the address space, even for the subarray, overflows the machine limit very soon. Another issue for representing the real life data by multidimensional arrays is that it creates a problem of high degree of sparsity and need to be compressed. It is therefore desirable to develop techniques that can access the data in their compressed form and can perform logical operations directly on the compressed data. In this research work we propose a data structure using the idea of EKMR and Traditional Extendible Array, namely Extendible Karnaugh Array (EKA) to represent the multidimensional data. The scheme has the intuitive propensity against the essential problem of address space overflow as well as it can be extended in any direction during run time. Moreover, we present a compression scheme for EKA to facilitate data access in compressed form. We evaluate our proposed scheme by comparing for different retrieval and extension operations with the Traditional Multidimensional Array (TMA). Our experimental result shows that the EKA scheme has a significant delay on the occurrence of address space overflow without any performance penalty. Furthermore, we find that range of usability of the compression scheme is independent of length or number of dimension. And it is better to use compressed EKA rather than uncompressed EKA for representing sparse data sets which needs range retrieval frequently.

Contents

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

Introduction

1.1 Introduction

There are few classes of data structures which are as well understood or as extensively used as arrays. It is quite often for scientific, statistical and engineering applications to have computation on large multidimensional arrays for modeling and analyzing scientific phenomena [1,2]. The strong need to handle large scale data proficiently has been promoting comprehensive research themes on organization or implementation schemes for multidimensional arrays on computer memory or secondary storage.

The fast random accessing capability of multidimensional arrays is a fascinating characteristic that enables various statistical computations including aggregation to be performed efficiently [3,4,5]. But this capability depends on the fact that the size of each dimension should be fixed so that a simple addressing function can be used to access an arbitrary element of the array. However, in many of the Multidimensional Online Analytical Processing (MOLAP) application data size grows incrementally. To represent those data, such kind of multidimensional arrays go through a serious problem namely extendibility; when a new data value is added, size extension along the corresponding dimension is necessary and this implies reorganization of the entire array elements. An array includes the necessity of dynamic extension of -

- Adding a new value to an existing dimension, for example a new product or a new city. This corresponds to adding a row/column in the corresponding dimension in the hierarchy.
- Giving a new level of aggregation for MOLAP, for example adding the aggregation on quarters to the time hierarchy.
- Addition of a totally fresh dimension such as age of customers.
- Modification of the definition of particular elements in the hierarchy. This usually happens when a different classification method is applied. For example age groups are defined in intervals of 5 years instead of 10 years.

The problem suffered by conventional array can be solved using extendible array model. An extendible array can be extended in any dimension without any repositioning of formerly stored data [6,7]. Such advantage makes it possible for an extendible array to be applied into wide application area where required array size cannot be predicted before and / or can vary dynamically during operating time of the system. The range in which the linearized array elements map is called address space – which depends on the length or number of dimension of array. When both the parameters are large, the address space becomes so large that it overflows conventional data types [8,9]. In this research work, we are going to propose a basic extendible data structure for handling large multidimensional data sets having the facility of managing the address space overflow.

1.2 Problem Statement

There are many existing array systems to represent multidimensional data such as Traditional Multidimensional Array (TMA) [10,11,12], Extended Karnaugh Map Representation (EKMR) [11,13], Traditional Extendible multidimensional Array (TEA) [7,14,15], Axial Vector Extendible array[16,17]. Besides these, there are some other extendible arrays such as Flexible Resizable Array [18,19] or Index Tree Extendible Array[20].

TMA or EKMR is a good storage for storing multidimensional data but one serious drawback is that they are not dynamically extendible. To insert a new column value in the TMA or in EKMR the total reorganization of the data in array is necessary. The idea of extendible array solves the problem of extendibility. However, extendible arrays use a concept of subarray. Extendible arrays, in fact, are combination of subarrays. If the array is *n* dimensional then the subarray is *n*-1 dimensional in many of the extendible array like TEA, Axial Vector array or Flexible resizable array. Even the subarray is n-dimensional in Index Tree array.

An *n* dimensional Array $A[l_1, l_2, .., l_n]$ is an association between *n*-tuples of integer indices and the elements of a set of *E*, whatever the domain of *E.* The set of continuous memory locations into which the array maps is denoted by *A*[0 : D], where D = $(\prod_{i=1}^{n} l_i) - 1$ and the size of the array is denoted by S = D×k, where k is the size of each cell. One more problem that suffered by all these above mentioned multidimensional array models is address space requirement. To allocate memory, consecutive memory location is required

for multidimensional array. But when the length of dimension l_i and number of dimension n of a multidimensional array is large then the address space overflows soon of the existing data types even for large configuration machines such as 64 bit machines. Hence it is impossible to allocate such a large size multidimensional array

Multidimensional arrays are good to store dense data, but most datasets are sparse which wastes huge memory because a large number of array cells are empty and thus are very hard to use in actual implementation. In particular, the sparsity problem increases when the number of dimensions increases. This is because the number of all possible combinations of dimension values exponentially increases, whereas the number of actual data values would not increase at such a rate. Many of the compression schemes like Compressed Row/Column Storage [21,22] or Chunk-offset Compression [23,24] already exist, but they are not suitable for extendible array models. So, efficient compression schemes are a strong requirement to store such sparse, incremental data for multidimensional data sets [25,26].

In brief, we are going to propose and evaluate a new and efficient implementation scheme of multidimensional extendible array model based on Karnaugh map [27], namely, Extendible Karnaugh Array (EKA), to manage the problem of extendibility without reorganization of data, overcome the address space overflow problem, and apply a suitable compression scheme on the model to have good compression ratio.

1.3 Scope

The basic operations of a standard data structure such as insertion, deletion, update and different categories of retrieval operations like existence check of record or item or entity, single key query, range key query are important and evaluated for traditional system implementations. Other important recent operations associated with multi-dimensional model of data under these domains are [17]:

- The data can incrementally grow over time by appending new data elements causing the length of dimension to be incremented dynamically.
- The datasets are mainly read-only for large amount of data. However, they may be subject to expansions in the bounds of the dimensions.
- The number of dimensions of the array may be increased or decreased. The array may grow (or shrink) by appending data for new time-steps.

The above scenarios are implemented in incremental update operation [3]. The increment operation, what we call extension, is efficient in the model because it increments without reorganizing the previous data. The increment operation will be analyzed along with the basic operations.

1.4 Objectives

MOLAP or various scientific applications use multidimensional array as a basic data structure to represent high dimensional data. This is because multidimensional array has an inherent facility to compute aggregation operation. Extendibility is an important requirement of those applications since data grows over time. Hence, an array model or realization scheme which can be extended over time is strong requirement of current era.

Therefore main objective of this research topic can be summarized as –

- Devise an implementation scheme for basic multidimensional array operations and to support incremental update operations.
- The TMA, TEA and EKMR suffer from the address space overflow problem. That is if the length of dimension and number of dimension become large then the coefficient values reaches the machine limits very quickly and overflows. Design will ensure the delay of overflow situation because of the division of subarrays.
- Provide the quick random accessing capability for different element search queries.
- Maintain the superiority of various array operations like point key, single key or range key query over TMA or other extendible array.
- Analyze performance of the proposed scheme on sparse array.
- Devise a suitable compression technique based on the proposed implementation model for representing a sparse array. And also analyze the performance and usability of the compression method.

1.5 Organization of the Thesis

- **Chapter II** presents Literature Review that describes some of the prominent array organization and realization scheme that are already exists. Here some of the on hand high dimensional data compression methods will be described.

- **Chapter III** proposes a new extendible array model based on Karnaugh map called as Extendible Karnaugh Array (EKA). It also explains the basic array operations like insertion, deletion, retrieval, extension etc. over the proposed EKA model.
- **Chapter IV** illustrates the details of compression method applied over proposed scheme.
- The experimental outcomes of different array operations over the EKA are discussed in **Chapter V.** It also presents the compression performance applied over EKA.
- The future direction of work on the proposed model and the conclusive words about the model are outlined in **Chapter VI.**

CHAPTER II

Literature Review

2.1 Introduction

Recently, multidimensional arrays are becoming important data structures for storing large scale multidimensional data; e.g., in statistical databases or MOLAP databases [20,28]. For analyzing purpose, scientific applications very often use multidimensional arrays to model high dimensional data. The solid demand of those applications leads novel researches on organization or implementation schemes for multidimensional arrays on computer memory or secondary storage.

2.2 Basic Terms and Notations

Multidimensional Array

An Array $A[l_1,l_2,\ldots,l_n]$ is an association between n-tuples of integer indices $\langle j_1, j_2, \ldots, j_n \rangle$ and the elements of a set of E such that, to each n-tuples given by the ranges $0 \le j_1 < l_1,\ 0 \le j_2 < l_2, \ldots,\ 0 \le j_n < l_n$ there corresponds an element of E. The domain from which the elements are chosen is immaterial and we make the assumption that only one memory location need be assigned to each n-tuples. Each array may be visualized as the lattice points in a rectangular region of n-space. The set of continuous memory locations into which the array maps is denoted by $A[0{:}D]$ where $D = (\prod_{i=1}^{n} l_i) - 1$.

Addressing Function

Any element in the multidimensional array is determined by an addressing function as follows,

$$f(x_n, x_{n-1}, x_{n-2}, \ldots, x_2, x_1) = l_1 l_2 \ldots l_{n-1} x_n + l_1 l_2 \ldots l_{n-2} x_{n-1} + \ldots + l_1 x_2 + x_1 \quad \ldots\ldots\ldots (2.1)$$

Conventional storage of multidimensional arrays is done by linearization. In the two dimensional case, the linearization may be done by rows or by columns. But in general,

for *n*-dimensional array there are n! possible linearization orders according to the possible ordering of the dimensions.

Coefficient Vector

The coefficients of the addressing function namely ($l_1 l_2 .. l_{n-1}, l_1 l_2 .. l_{n-2}, ..., l_1$) is referred to as coefficient vector and stored during the construction time. Hence the addressing function can be computed very fast at the element access time.

Length of Dimension

Each of $l_i\,(1 \le i \le n)$ is determined as length of dimension *i* of a multidimensional array.

Subarray

A subarray S*A*[l_1,l_2,…,l_{n-1}] is an association between *n*−1 tuples of integer indices $\langle j_1, j_2, \ldots, j_{n-1} \rangle$ and the elements of a set of *E* such that, to each *n*-1 tuples given by the ranges $0 \le j_1 < l_1,\ 0 \le j_2 < l_2, \ldots,\ 0 \le j_{n-1} < l_{n-1}$ there corresponds an element of E. The set of continuous memory locations into which the array maps is denoted by $SA[0 : D]$ where $D = (\prod_{i=1}^{n-1} l_i) - 1$.

Segment

For an *n*−1 dimensional subarray, segment is a part of subarray of dimension *n*−2. That is a subarray S*A*[l_1,l_2,…,l_{n-1}] can have l_{n-1} segments each which size is $l_1 \times l_2 \times \ldots \times l_{n-2}$.

2.3 The Realization of Multidimensional Array

Multidimensional array has an inherent facility of random accessing – the reason of becoming the most popular. But capability demands the length, and number of dimension to be fixed – which leads problem of dynamic extension. There are many data structures already exist to represent multidimensional data. Some of them are static in nature and some are dynamic – i.e. resizable without reorganizing the already allocated data. Some of the well-known and prominent data structures are discussed in this section.

2.3.1 Traditional Multidimensional Array (TMA) [10,11,12]

Traditional Multidimensional Array (TMA) is a scheme for representing multidimensional data. The TMA represents *n* dimensional data by an *n* dimensional array. The key to the structure of arrays resides in the familiar coordinate system, which pictures an *n*-

dimensional array as being imbedded in the positive orthant of n-dimensional space, with array positions lay on the lattice points. An illustration of 3 dimensional TMA of dimension length 3×4×5 is given in Figure 2.1.

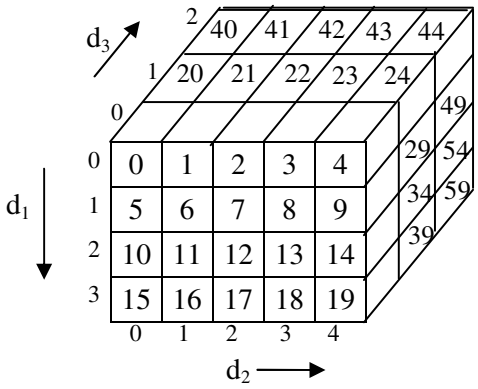


Figure 2.1: A three dimensional TMA with length of dimension 3×4×5.

In the TMA scheme, a three dimensional array of size 3×4×5 can be viewed as three 4×5 two-dimensional arrays. An element (x_n, x_{n-1}, …, x_1) in an n dimensional Traditional Multidimensional Array of size [l_n,l_{n-1}, …, l_1] is allocated on memory using an addressing function like eq. (2.1)

We already know from the definition of addressing function that there are n! possible linearization orders for an n-dimensional array. Storage by linearization allows extension without any movement of existing elements only in one of the dimensions. For example we can readily extend the 3D TMA of Figure 2.1 only in third dimension, but in other case reorganization is necessary for already allocated cell.

2.3.2 Extended Karnaugh Map Representation (EKMR)

A basic array representation scheme named Extended Karnaugh Map Representation (EKMR) is proposed in [11,13]. In this scheme, an n-dimensional array is represented by a set of 2 dimensional arrays. The idea of the EKMR scheme is based on the Karnaugh map (K-map). Consider a 3 input K-map and its corresponding EKMR(3) in Figure 2.2. The analogy between the EKMR(3) and the 3-input Karnaugh map is that the index variables i, j, and k correspond to the variables X, Y, and Z respectively. Here, index variable i is used to indicate the row direction and the index variable j is used to indicate the column direction. When n = 1 and 2, the TMA and the EKMR schemes are the same.

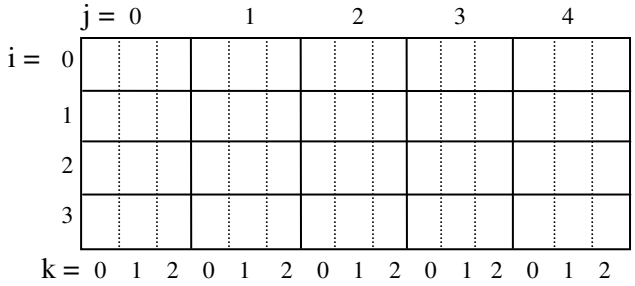


Figure 2.2: EKMR representation of a 3 dimensional array, EKMR(3).

Let A[s][r][p][q] be a 4-dimensional array based on TMA with size $s \times r \times p \times q$. A′[$u$][$v$] be the corresponding EKMR(4) of size $(s \times p) \times (r \times q)$. Let A and A′ are presented as row major order, so location of a cell for given subscripts in A and A′ can be calculated as

$$L_A(l, k, i, j) = p \times q \times r \times l + p \times q \times k + q \times i + j$$

$$L_A'(i', j') = r \times q \times i' + j'$$

And the mapping function from L_A to L_A' can be defined as follows:

$$L_A(l, k, i, j) \rightarrow L_A'(i', j'),$$

$$\text{where } \begin{cases} i' = i \times s + l \\ j' = j \times r + k \end{cases}$$

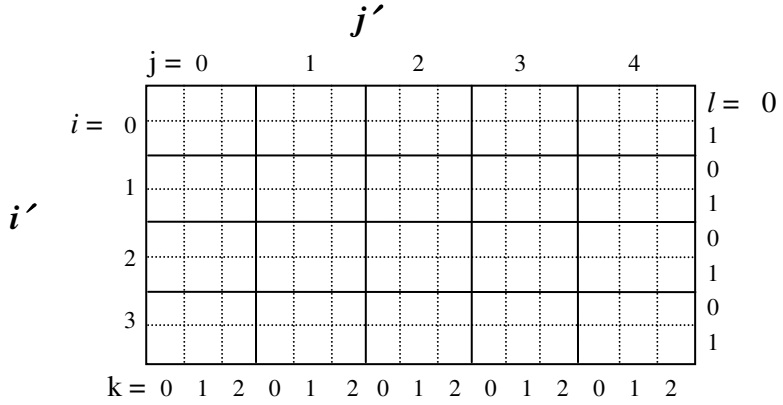


Figure 2.3: EKMR representation of a 4 dimensional array, EKMR(4).

Consider an array A[2][3][4][5] represented as a TMA(4). The corresponding EKMR(4) of array A is shown in Figure 2.3. The EKMR(4) is represented by a $(2 \times 4) \times (3 \times 5) = 8 \times 15$ two-dimensional array. The basic difference between TMA(4) and the EKMR(4) is the placement of elements along the direction indexed by k, and l. The relative position makes the fundamental difference when using EKMR as array representations.

Based on the EKMR(4), the EKMR(n) can be represented by m^{n-4} EKMR(4) and a one-dimensional array X with a size of m^{n-4} are used to link these EKMR(4).

2.3.3 Traditional Extendible Array (TEA) [7,14,15]

The Traditional Extendible Array (TEA) is another representation of multidimensional array. It has the property that the indices of the respective dimensions can be arbitrarily extended without reorganizing previously allocated elements. Following is a short description of TEA.

Extendible arrays are combination of subarrays. If the array is *n* dimensional then the subarray is *n*-1 dimensional. It has three types of auxiliary tables namely history table, coefficient table and address table. For each dimension these tables exist. There is a history counter that counts the construction history of the sub arrays. Address table contains the first address of the subarray, history table contains the construction history of the subarrays. Coefficient table holds the coefficient of the *n*-1 dimensional subarrays. The coefficient vector is *n*-2 dimensional. The extendible array can be extended in any direction in any dimension only by the cost of these three auxiliary tables.

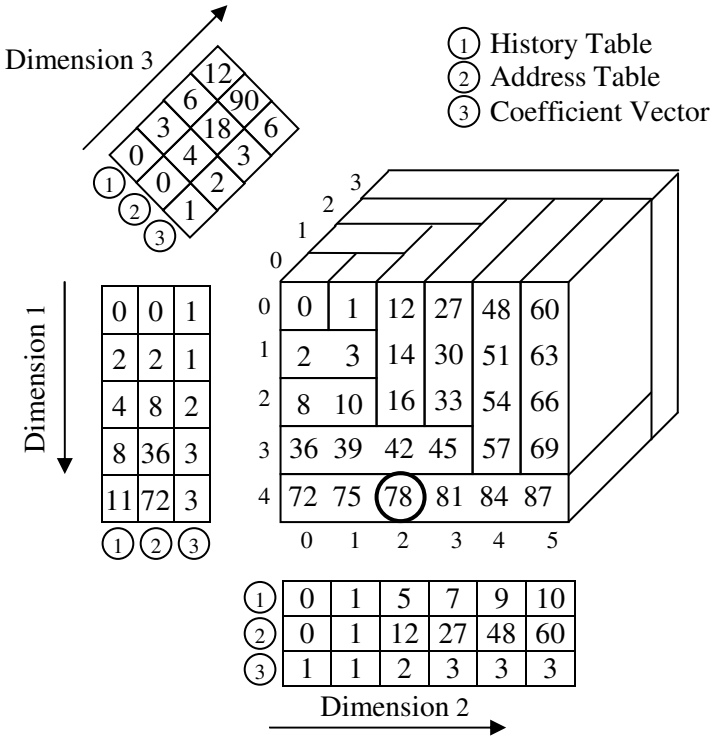


Figure 2.4: A Three dimensional Traditional Extendible Array.

The accessing of the elements of an extendible array is completely different from the conventional multidimensional array. Using these three kinds of auxiliary tables, the address of an array element can be computed as follows. Consider the element ⟨4, 2, 0⟩ in Figure 2.4. Compare $H_1[4] = 11$, $H_2[2] = 5$ and $H_3[0] = 0$. Since $H_1[4] > H_2[2]$, $H_1[4] > H_3[0]$, it can be proved that the element ⟨4, 2, 0⟩ is involved in the extended subarray S beginning from the address $L_1[4] = 72$. From the coefficient vector of $C_1[4] = 3$, the offset of element ⟨4, 2, 0⟩ from the first address of S is computed by $3\times2 + 0 = 6$, the address of the element is determined as $72 + 6 = 78$.

Storage costs of the history tables and the address tables of an n dimensional extendible array are both $O(n)$. On the other hand, the storage cost of the coefficient tables is $O(n^2)$, because a coefficient vector consists of $n-2$ constants and the total size of the coefficient tables is proportional to $n(n-2)$. In an environment where both of n and the dimension sizes are large, to suppress the size of these auxiliary tables is very important in order to place them on main memory and make them work as an index for the array elements placed on secondary storage.

2.3.4 Axial Vector Array [16,17]

In axial vector method there is record for each dimension called axis vector. Each element of the vector stores necessary information (starting index of the dimension, starting address of the subarray, multiplicative coefficients, and memory pointers) to retrieve an element correctly.

In this approach the sequence of the two consecutive extensions along the same dimension, although occurring at two different instances, is considered as an uninterrupted extension of that dimension and handled by only one expansion record entry in the axial-vector. Therefore number of element in an axial vector is always less than or equal to the number of indices of the corresponding dimension.

Figure 2.5 shows the extension of a three dimensional array A of initial size 4×3×1, and corresponding axial vectors. At the very first, the array extended twice in third dimension d_3, then extended in d_2, d_1, and finally in d_3 again.

To correctly compute the linear address of k-dimensional index $\langle i_1, i_2, \ldots, i_k \rangle$, determine which of the records in axial vector $\Gamma_1(z_1)$, $\Gamma_2(z_2)$. . . $\Gamma_k(z_k)$, has the maximum starting

address. Here, the index z_j is the highest index of the axial-vector where the expansion record has a starting index less than or equal to i_j .

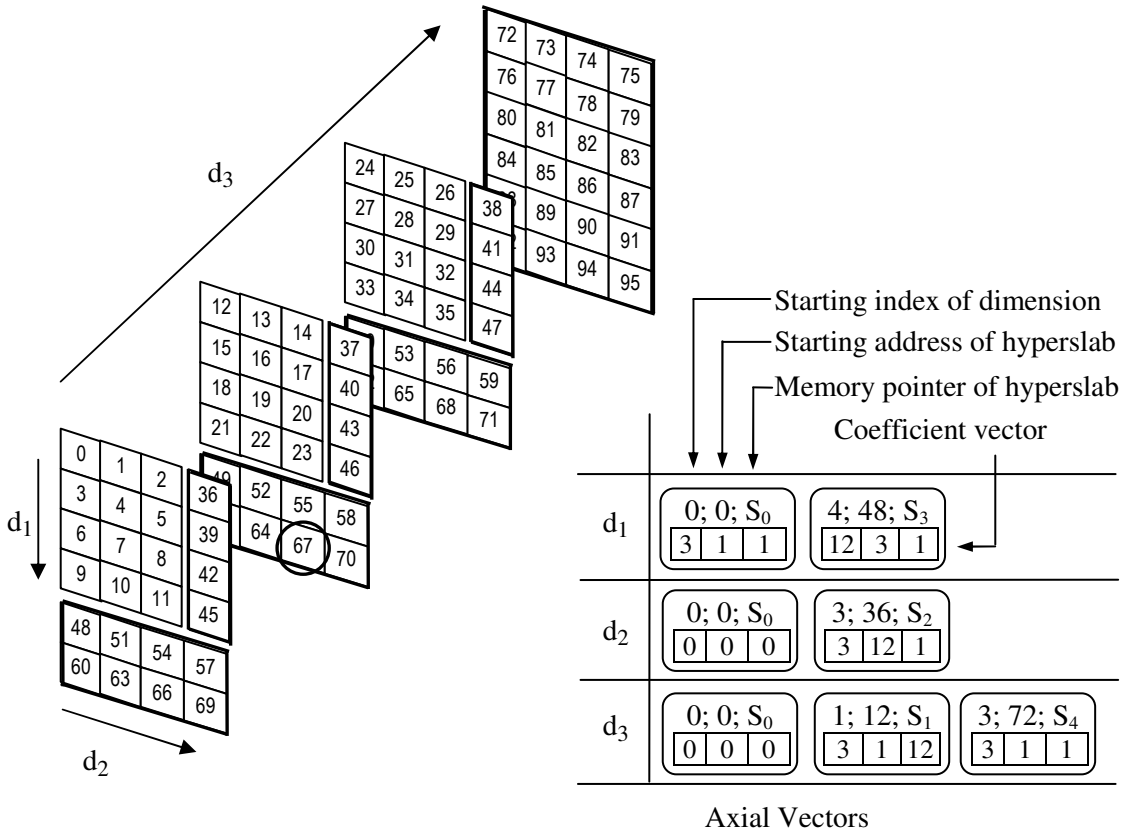


Figure 2.5: A 3-dimensional extendible array along with axial vectors.

For example, suppose we desire the linear address of the element A[5,2,1], we first note that $z_1 = 1$, $z_2 = 0$, and, $z_3 = 1$. Now $\max(\Gamma_1(1), \Gamma_2(0), \Gamma_3(1)) = \max(48, 0, 12) = 48$, maximum dimension is d_1, and starting index of d_1 is 4. So linear address is = 48 + (5-4)*12 + 2*3 + 1*1 = 67 (encircled), where 12, 3, and 1 are multiplicative coefficients.

2.3.5 Chunking of Array [1,29]

To address the problems faced by applications that do not perform well with traditionally ordered arrays on disk, The data management libraries that support storage of multidimensional arrays on disk with the elements arranged in subarray chunks rather than in the traditional ordering is important. Figure 2.6 shows a 3D array having multidimensional chunks. This allows efficient assembly of subarrays in multiple dimensions.

A single array that can be divided into chunk is known as chunking that can be stored contiguously on disk by storing the chunks of the array according to a predefined ordering on the chunks. The chunk number can be used to compute the correct offset of the chunk from the beginning of the array at run time. A single array may have chunks of different sizes, since real-world problems come in a variety of sizes that do not guarantee an even distribution of the processor. The chunks can be stored on disk in a packed or unpacked fashion. In the unpacked case we can assume all array chunks on disk occupy the same number of bytes as the largest chunk, to simplify the calculation of the offset of a chunk. With this approach, the smaller chunks result in unused space on disk. Chunks can be stored on disk packed together with no waste space if additional computations at run time are performed to calculate precise offsets to packed chunks, or by using a directory structure of points that can be persistent or created on demand.

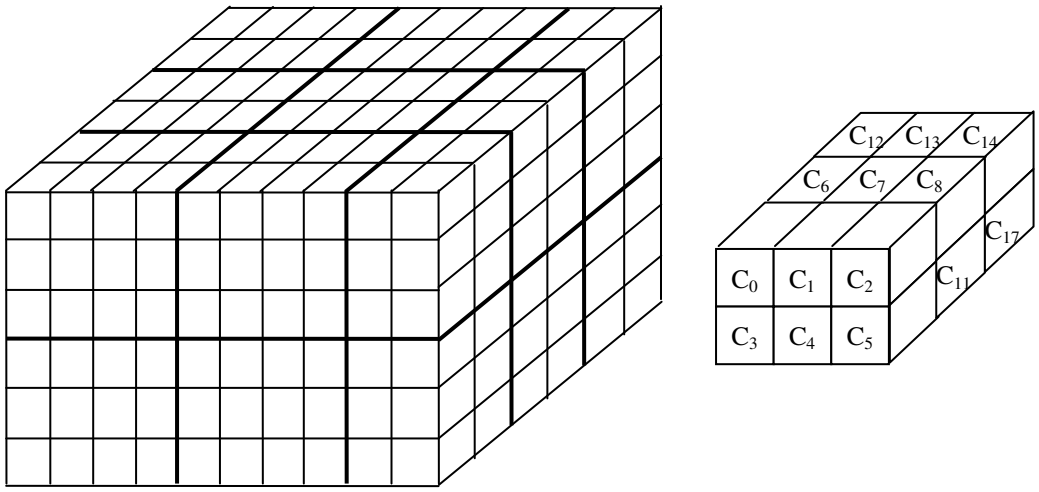


Figure 2.6: A 3-dimensional array partitioned into chunks.

When an application has multiple arrays that are logically related to each other and are accessed together, it is common to assign corresponding chunks the same area. The equivalent concept from the database community is that of clustering data that is used together should be placed together on disk. This is referring to as physical schema as interleaving chunks. Figure 2.7 shows how the 16 chunks of each array X and Y are ordered on disk using an interleaved strategy. Note that it is not necessary that this array all be of the same data type or even the same size.

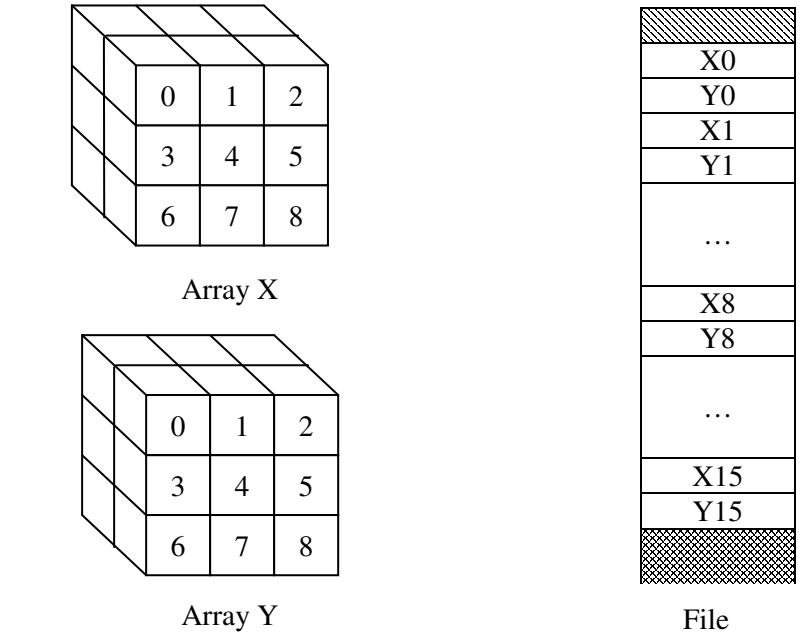


Figure 2.7: Arrays X and Y are stored using interleaved chunks.

2.3.6 Flexible Resizable Array [18,19]

This is a variant of TEA. Unlike a TEA, in this organization it is possible to insert in the midst of the array. However such insertion would influence the logical location of other array elements; for example the location of the element (A) in Figure 2.8 changes from ⟨1, 1⟩ to ⟨2, 1⟩. Note that the logical locations of the array elements would be changed by these insertions, but their physical locations would not be changed. Therefore the offset computation described in Section 2.3.3 cannot be applied as it is. Here, the errors in offset calculation are compensated to get the correct physical location by an efficient mapping mechanism.

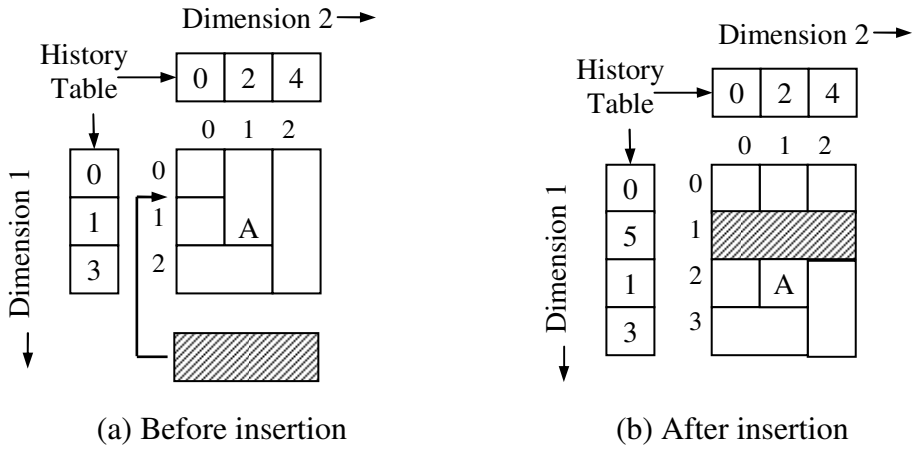


Figure 2.8: Insertion of a subarray in the midst of a 2-dimensional TEA.

Along with auxiliary tables of TEA, this scheme uses *bitmap table* for each dimension that consists of a set of pairs (history value, bit sequence). Each history value in this set is used for selecting the bit sequence to be used in calculating the extension compensation. The bit sequence holds information of insertion positions of the dimension and is used for determining the number of positions to be taken into account for compensation. And bit 1 denotes that the position should be compensated.

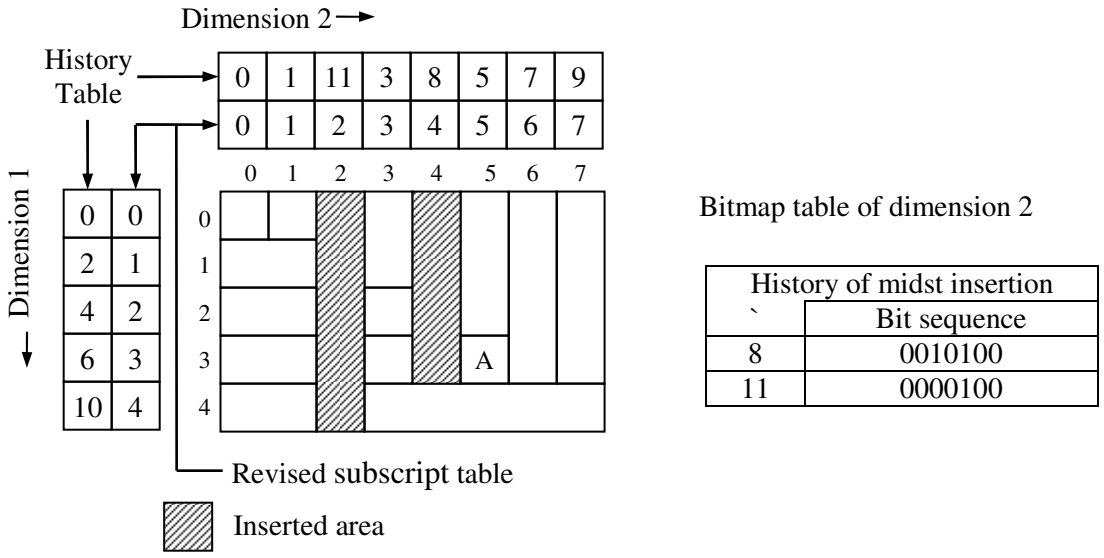


Figure 2.9: A 2-dimensional Flexible Array with revised subscript and bitmap table.

Figure 2.9 shows the bitmap of the second dimension. For the element A(3, 5) in Figure 2.9, the process to obtain the extension compensation value of the second dimension is presented here. The principal subarray of the element A has the history value 6 of the dimension 1. Compensations for the other dimensions, in this case dimension 2, are necessary. So the specified subscript 5 of the dimension 2 should be compensated and this is done as follows:

- Find the least value in bitmap table that exceeds the history of principal subarray, namely 6. In this case, the history value 8 would be found. Hence the bit sequence 0010100 will be used.
- Since the revised subscript of the subordinate subarray of A in the dimension 2 is 5, the total number of set bits is counted up to the bit 5 of the bit sequence 0010100; the right most bit being the bit 0. The total number is 2, so the extension compensation value of the second dimension is concluded to be 2.

In this way, for n-dimensional array compensation value δ_k is calculated, and the compensated coordinate $\langle i'_1, i'_2, ..., i'_n \rangle$ of an element e will be computed as $i'_k = i_k - \delta_k$ for each dimension k ($1 \leq k \leq n$). The user-specified coordinate of subscripts before compensation is used to determine the maximum history value, hence the principal subarray of e, and the compensated coordinate is used to determine the correct offset in the principal subarray.

2.3.7 Index Tree Extendible Array [20]

In this approach a tree-based index is used to keep track of the growth of the array in any dimension and even allow adding of new dimensions. An extension of a k-dimensional array A along dimension i is viewed as appending a k-dimensional subarray A^S to it along the ith dimension. The ranges of A^S are identical to those of A along each dimension except for dimension i whose range depends on the size of the extension. The length l_i, of dimension i is called as the range of dimension i.

The index is a search tree (can be implemented as a B-tree) based on a compound key (D, L) where D represents the dimension number and L the new range that this dimension achieves after extension. Each key points to an associated record containing some information about the extension as described below.

Each time the array is extended, a new key will be inserted into the search tree indicating the dimension that has grown and the new range value for that dimension. The general structure of the associated record is R = (r_0, r_1, r_2, .., r_{k-1}) where r_0 is the starting address of the extension subarray, and the other r_i-s indicate the maximum lengths along the non-extended (other k-1) dimensions at the time of extension. When an extension along dimension i is performed, the value of r_j for $j<i$ indicates the maximum range along dimension j, and for $j \geq i$ it indicates the range for dimension $j + 1$.

In general, for a k-dimensional array, an extension along dimension j with extension size s causes an insertion of a key K, and associated record R can be summarized as below shown in Table 2.1.

Table 2.1. Summary of insertion key and associated record of B-tree.

	Key, K	**Associated Record, R**
If $j \leq k$, i.e. extension on any existing dimension	$(j, l_j + s)$	$(\sum_{i=1}^{k} l_i , l_1, l_2, ..l_{j-1}, l_{j+1} ,.., l_k)$
If $j > k$, i.e. a new dimension is introduced	$(k+1, s)$	$(\sum_{i=1}^{k} l_i , l_1, l_2, ..,l_k)$

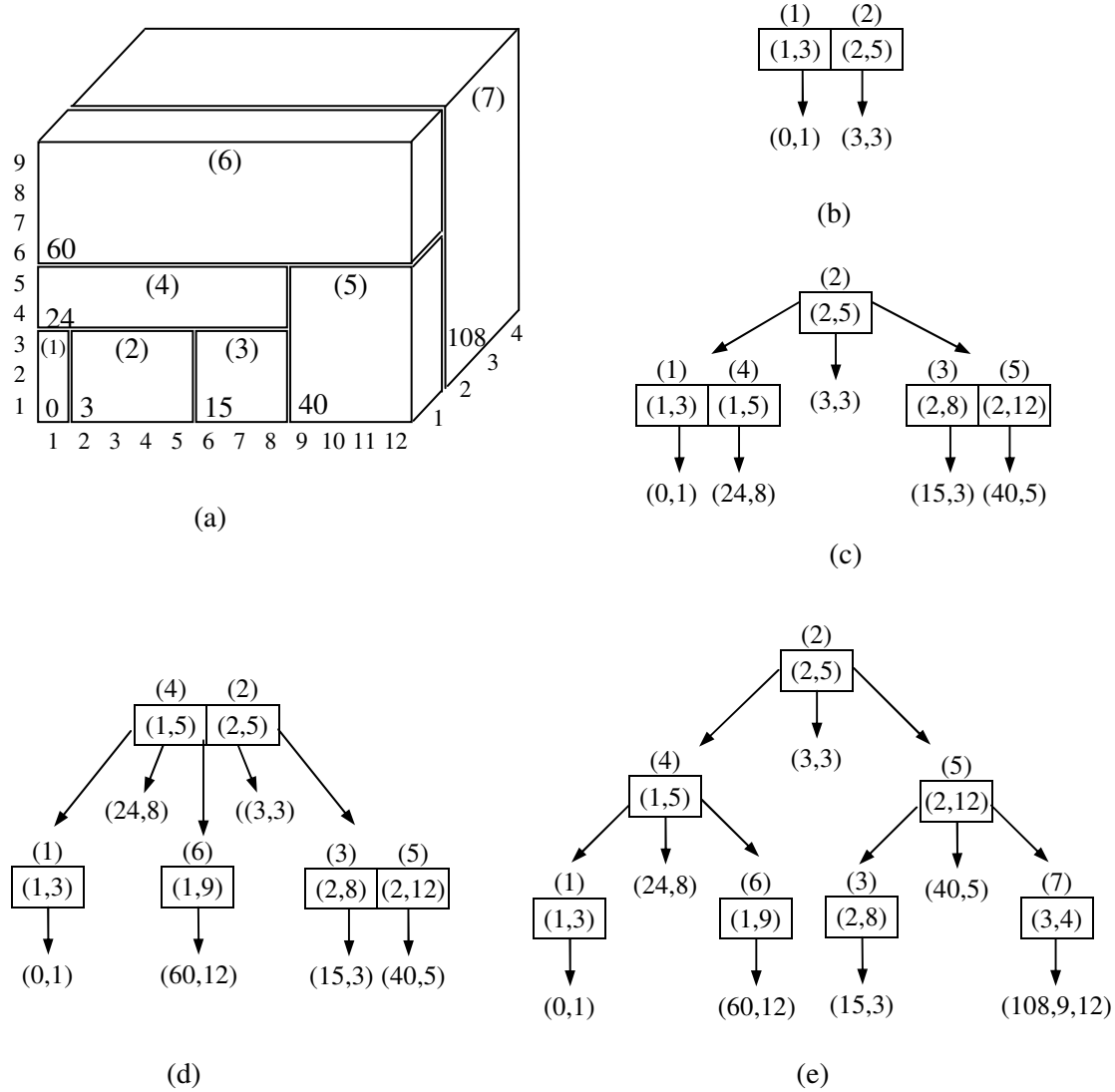


Figure 2.10: Realization of 3-dimensional Index Tree extendible array.

Consider a two dimensional array of 3 rows and 1 column, starting at address 0. Assume the rows are dimension 1 and the columns dimension 2. The initial insertion includes the key K = (1,3) and associated record R = (0, l). The first extension is on dimension 2 to 5 columns, key inserted is K = (2,5) and the record is R = (3,3)

indicating the starting address of the extension subarray is 3 and the non-extending dimension range is also 3. After that the array is extended to 8 columns, so the tree will have a key K = (2,8) and record R = (15,3). Next extension is to 5 rows with K = (1,5) and R = (24,8). Then extension is made to 12 columns causes the insertion of K = (2,12) and R = (40,5). Extension to 9 rows causes insertion of K = (1,9) and R = (60,12). Suppose, now a new dimension to the array added so it becomes a three dimensional array and extend this new dimension to a range value of 4, the key inserted in this case will be K = (3,4) and R = (108,9,12). The K and R entries for the extendible array are illustrated in Figure 2.10. The B-tree in that example can hold a maximum of 2 keys per node.

To find an element *e* of index (4, 9, 3) iteratively search the B-tree of Figure 2.10 until desired information is found. Start with a search for the key (1,4), the first key found in the B-tree larger or equal to it is (1,5) having associated record (24, 8) - means the range of dimension 2 is only 8, smaller than the required 9. The search continues along dimension 2, the key searched for is (2, 9) and the key found is (2,12) with associated record of (40, 5) is corrected to (40, 5, 1) due to a missing component for 3rd dimension. The range for dimension 3 is still too small than the required 3. Finally, for dimension 3, the key searched for is (3, 3) and the key found is (3,4) with an associated record (108,9,12). All components are larger than the corresponding ones, so we conclude that the element *e* is found in the extension subarray represented by (3, 4).

2.4 Compression Schemes for High Dimensional Data

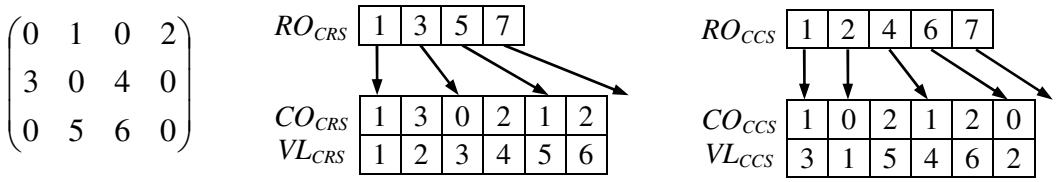
Multidimensional array are the basic data structure used in many applications such as MOLAP. But in many cases, they are found to be sparse in nature – i.e. many of the array cells contain null values and consume unnecessary space. So it is important to design a technique, “The Compression”, to store such arrays. Some common compression methods are reviewed here.

2.4.1 CRS/CCS Schemes [21,22]

Let, a two-dimensional sparse array is given. The Compressed Row/Column Storage (CRS / CCS) scheme using one one-dimensional floating point array VL and two one-

dimensional integer arrays RO and CO to compress all of the nonzero array elements along the rows (columns for CCS) of the sparse array. Array RO stores information about the nonzero array elements of each row (column for CCS). The number of nonzero array elements in the *ith* row (*jth* column for CCS) can be obtained by subtracting the value of RO[i] from RO[i+1]. Array CO stores the column (row for CCS) indices of nonzero array elements of each row (column for CCS). Array VL stores the values of nonzero array elements. The base of these three arrays is 0.

An example of the CRS/CCS schemes for a two-dimensional sparse array is given in Figure 2.11(a) that shows a 3×4 two-dimensional sparse array. Figure 2.11(b) and Figure 2.11(c) show the CRS/CCS schemes, respectively. For four or higher dimensional sparse arrays based on the TMR scheme, more one-dimensional integer arrays are needed.



(a) A sparse array (b) The CRS Scheme (c) The CCS Scheme

Figure 2.11: The CRS/CCS schemes for a two-dimensional sparse TMA.

2.4.2 Chunk Offset Compression [23,24]

In this scheme the large multidimensional arrays are broken into chunks for storage and processing. Consider an n-dimensional array A, whose size is $l_1 \times l_2 \times l_3 \times \ldots \times l_n$, the chunks can be formed by breaking each l_i into several ranges. Within A, two positions are in the same chunk if and only if, in every dimension, they fall within the same range. In memory or disk, values within a chunk are stored consecutively. Elements in a chunk are arranged according to the pre-specified order of dimensions. In this compression technique, the pairs of (*offset value in the chunk, fact data value*) are physically stored in secondary storage only on nonempty elements in a chunk. This set of pairs is sorted in the order of the offset values. Figure 2.12(a) shows a 3 dimensional array partitioned into 36 chunks each of which is 3×3×3 (Figure 2.12(b)). The details of a chunk with 8 data values and offset within the chunk are shown in Figure 2.12(c), and Figure 2.12(d) displays memory

or disk arrangement of that chunk. Note that the chunks which have no nonempty elements are not physically allocated in the secondary storage.

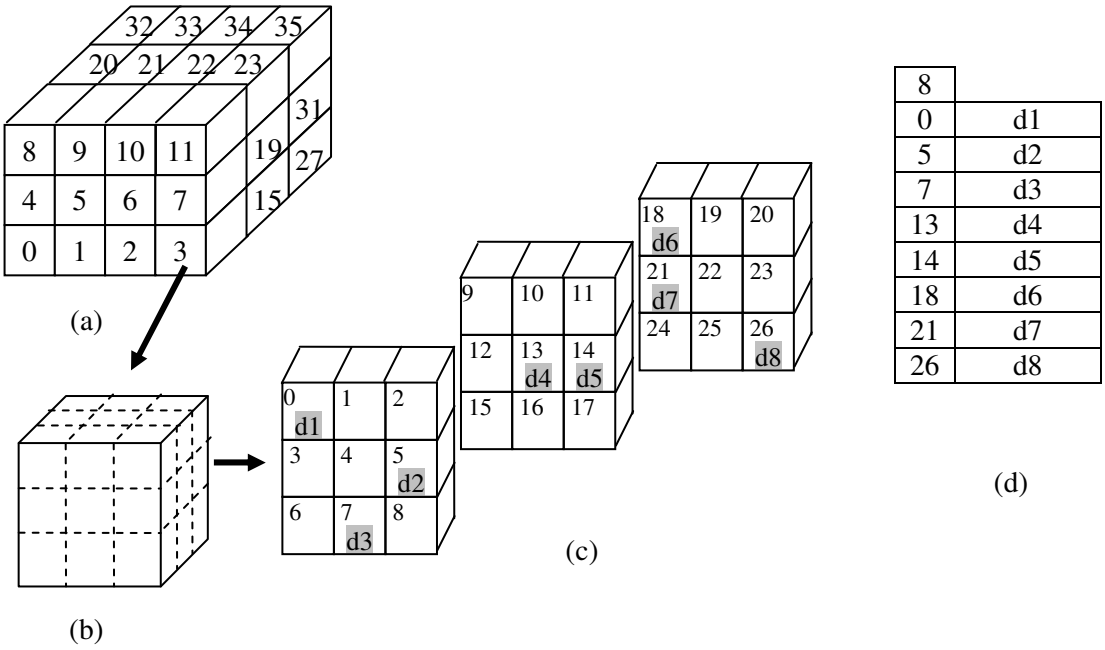


Figure 2.12: A 3-dimensional array stored as chunk-offset compression.

2.4.3 History Offset Compression [6,10]

The history offset compression scheme is based on extendible array. In this technique, an element is specified using the pair of history value and offset value of the extendible array. Since a history value is unique in extendible array and has one to one correspondence with the corresponding subarray, the subarray including the specified element of an extendible array can be referred to uniquely by its corresponding history value *h*. Moreover, the offset value (i.e., logical location) of the element in the subarray can be computed by using the addressing function and this is also unique in the subarray. Therefore, each element of an *n*-dimensional extendible array can be referenced by specifying the pair (history value, offset value). Like Chunk-offset compression, the extended sparse subarray elements are stored in memory in sorted fashion.

2.4.4 Some Other Compression Schemes

In memory or secondary storage, array elements are stored in linear fashion. Some of the compression schemes described below are applicable only after the linearization of the array.

Bit Map Compression [30,31]

A bit map compression scheme consists of a bit map and a physical database which stores the non-constant values. The bit map is employed to indicate the presence or absence of non-constant data. The following example shows how the bit map compression scheme can be employed to implement a version of constant suppression.

Original data string

d1, c, c, d2, c, c, c, d3

Compressed data string

Bit map: 10010001.

Physical database: dl, d2, d3.

For the bit map compression method, the mapping mechanism must search the whole bit map for both forward and backward mapping. And thus, the access time for both forward and backward mapping is O(N), where N is the number of bits in the bit map or equivalently the number of elements in the database.

Header Compression [32]

The header compression scheme is shown below. The vector L, represents the uncompressed form of a database, in which the 0's are the constant to be suppressed and the V's are the unsuppressed values. Beneath the vector L is the list of counts which comprise the compression header, H. The odd-positioned counts hold accumulations of unsuppressed values; and the even-positioned counts hold the accumulations of zeros. The physical, compressed form of the data is represented by P.

L: V1V2000000000V3V4V5V6V700V8V9V10000

H: 2, 9, 7,11,10,14

P: V1 V2 V3 V4 V5 V6 V7 V8 V9 V1O

For the header compression method, the forward and backward mapping can be processed by binary searching on the header, H. Both of them require O(log s) time where s is the size of the header.

2.5 Discussion

All the array models presented in this chapter have some pros and cons. Since TMA and EKMR have pre-specified length and dimension, they are good for random accessing. But they suffer in case of dynamic extension. The TEA, Axial Vector array, and Flexible resizable array are good for dynamic extension. TEA and Axial Vector array provides extension at the boundary where as Flexible array allows even in the middle of the array. But they all have a concept of subarray which is always $n-1$ dimensional. For large value of length for each dimension or for large number of dimension value of offset grows exponentially and overflows the address space. In case of Chunking of Array or Index Tree array the subarray is n dimensional, so they also suffer from address space problem.

Classical compression schemes have some limitations in compressing data. Like Bitmap and its derivatives such as Header compression provide good performance in terms of removing long runs of constants, but they have a poor forward and backward mapping capability. Also, these methods can't be used on dynamic database environment where additions and deletions may be required. The scheme Compressed Row Storage (CRS) or Chunk Offset compression are effective for compressing large sparse arrays. But still they cannot be applied on extendible databases. So, it is important to design a compression technique that will be better than these classical compression techniques. The scheme should be efficient enough so that operation can be done over the compressed data.

Though, there are a lot of research has been done on array model, but only a few researches have been made on dynamic array organization even hardly any on overflow situation. Hence we propose a dynamic array model which will outperform over TMA as well as overcome the overflow scenario. The detail of the proposed scheme is presented in the next chapter.

CHAPTER III

The Extendible Array Representation using Karnaugh Map

3.1 Introduction

Conventional schemes for storing arrays do not support easy dynamic extension of an array. The conventional storage allocation scheme for arrays is either row major or column major ordering. Though the allocation technique provides optimal storage utilization but the extension of the dimensions lacks in all except single dimension. Such asymmetry in extendibility is not inevitable. However, such kind of multidimensional arrays go through following two important problems:

(i) The size of the multidimensional array is not dynamically extendible; when a new data value is added, size extension along the corresponding dimension is necessary and this implies reorganization of the entire array elements.

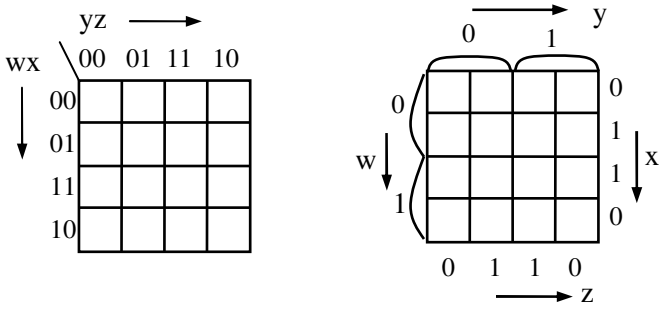
(ii) Another problem with the multidimensional array is address space requirement. To allocate memory, consecutive memory location is required for multidimensional array. But when the length and number of dimension of a multidimensional array is large then the address space overflows soon.

It is devised schemes for multi dimensional storing arrays, which are readily extendible in all directions. An extendible array, however, does not store an individual array; rather, it is storing an array and all its potential extensions. The scheme is an *n* dimensional rectangular array that grows by adjoining blocks, which are subarrays of dimension *n-1*. Within which each subarray storage allocation is in row-major or Lexicographic order.

3.2 The Realization of Extendible Karnaugh Array

The idea of the proposed scheme, what is named is Extendible Karnaugh Array (EKA), is based on Karnaugh Map (K-map) [27]. Karnaugh maps are used to facilitate the simplification of Boolean algebra functions usually aided by mapping values for all possible combinations. Input values are arranged in Grey Code. Figure 3.1 (a) shows a 4

variable K-map to represent possible 2^4 combinations of a Boolean function. The variables (w, x) represent the row and the variables (y, z) represent the column to indicate the possible combinations in a two dimensional array.



(a) 4 variable K-map (b) Array representation

Figure 3.1: Realization of Boolean function using K-map.

The 4 variable K-map can be easily drawn plane as a two dimensional array which is shown in Fig. 1(b). Here each of the boundary line directs a dimension. The length of each of the dimensions is 2 for both Figure 3.1(a) and (b). This is because the Boolean variables are binary that causes the length to be 2.

Definition 3.1 (Adjacent Dimension): In array representation, the dimensions (or index variables) that are placed together in the Boolean function representation of K-map are termed as adjacent dimensions (written as *adj*(i) = j). The dimensions (w, x) are the adjacent dimensions in Fig. 1, i.e. *adj*(w) = x or vice versa.

3.3 The 4-Dimensional EKA Scheme

EKA represents the array as the combination of subarrays. Besides, it has three types of auxiliary tables namely *history table*, *coefficient table*, and *address table*. For each of the four dimensions these tables exist. These tables store the extension information and help the elements of the EKA to be accessed very fast.

The extension subarrays are further sub divided into number of segments. The number of segments determines the number of entries in the address table and is calculated from the length of adjacent dimension. When extension along a dimension is in progress, the extension subarrays are three dimensional, therefore segmented subarrays are always two dimensional for an EKA(4). We write EKA(*n*) to mean an *n* dimensional EKA.

There is a history counter that counts the construction history of the subarrays. *History table* contains the construction history of the subarrays. For each history, the *address table* contains the first address of the extended subarrays for the corresponding dimension. Since for each extension the subarrays are broken into segments, the address table, in fact, stores the first addresses of each segments of the subarray. Hence for a single subarray (or history value) the address table entry can be more than one.

Since EKA is a dynamic array, the coefficient vectors for different subarrays are distinct. So, to retrieve the array elements accurately these coefficient vectors are stored in *Coefficient table*. As each segment of the subarray is 2 dimensional hence in our model the coefficient vector becomes $\langle l_1 \rangle$ only. The EKA can be extended along any dimension dynamically during runtime only by the cost of these three auxiliary tables.

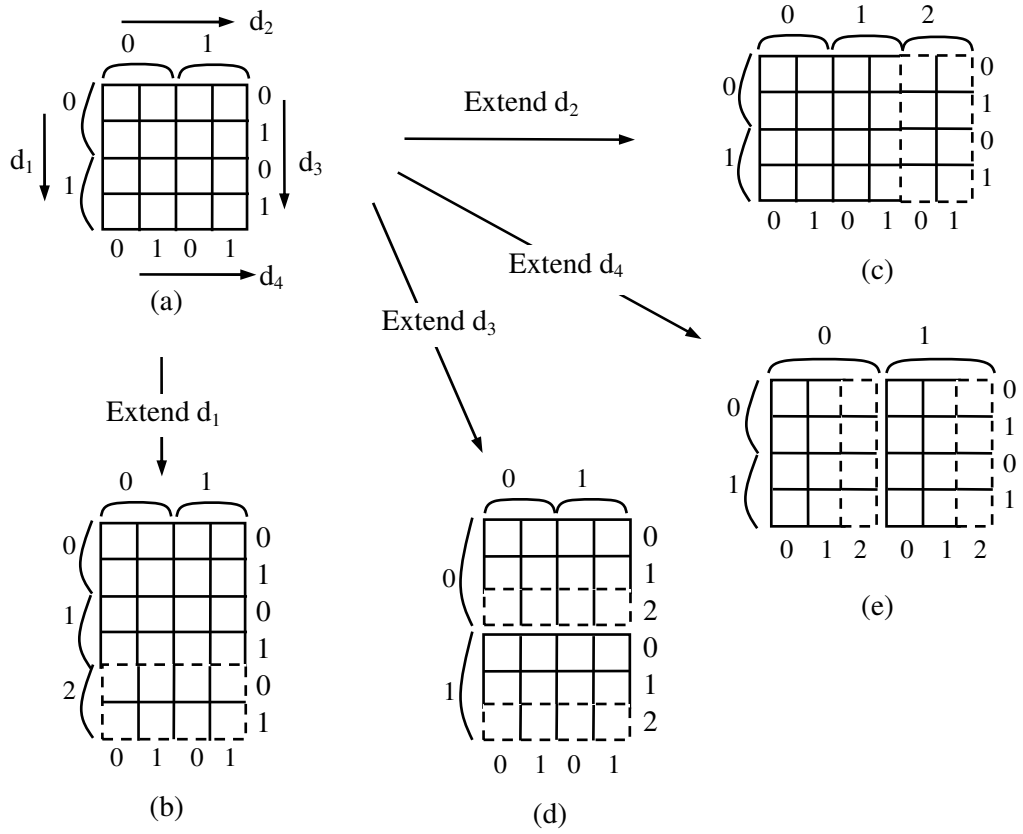


Figure 3.2: Logical extension of 4-dimensional EKA.

Figure 3.2 represents a four dimensional EKA, EKA(4). The dimensions are d_1, d_2, d_3 and d_4 and the size of the array is $[l_1, l_2, l_3, l_4]$ where l_i indicates the length of dimension d_i and subscripts varies from 0 to $l_i - 1$. In the current example $l_i = 2$. The dimension (d_1, d_3) and (d_2, d_4) are adjacent dimensions respectively. The logical extension in d_1 is shown in

Figure 3.2(b). The size of the extended subarray which is allocated dynamically is determined by $l_2 \times l_3 \times l_4$ (i.e. other 3 dimensions). The number of segments is the length of the adjacent dimension, $adj(d_1) = d_3$; In this case it is $l_3 = 2$. The size of each segmented subarray extended along dimension d_1 is determined by $l_2 \times l_4$. After extending along dimension d_1, the length of that dimension is incremented by 1. For each extension the corresponding auxiliary tables are maintained accordingly. Figure 3.2(c), 3.2(d) and 3.2(e) shows the extension realization along dimension d_2, d_3 and d_4 respectively.

3.3.1 Illustrative Example of EKA(4)

We have taken an EKA(4) as an example that is shown in Figure 3.3. We are going to extend it in different dimensions in the following way.

Figure 3.3(a) illustrates the initial setup of the scheme. The history counter is zero and the history tables contain one entry namely 0. The address tables contain first address which zero here. Each of the coefficient table entry is 1 since length of each dimension is 1. During the extension of d_1 and d_3 dimension size of the segment is $l_2 \times l_4$ which is a two dimensional array, and so coefficient vector is one dimensional. Hence, for our example we use l_2 as coefficient vector for d_1 and d_3 dimensions. Similarly, l_3 is used as coefficient vector for d_2 and d_4 dimension and coefficient table is maintained. When an extension along d_2 direction is done as shown in Figure 3.3(b), the history counter is increased by 1. The value of history counter is stored in the history table H_{d2}. The subarray size [l_1, l_3, l_4] is calculated and dynamically allocated; the values of first address are stored in address table A_{d2}; since l_3=1, C_{d2} stores this value. Figure 3.3(c) shows an extension along d_3 direction. Here the history counter incremented by 1 and this is stored in history table H_{d3}. The size of the subarray [l_1, l_2, l_4] is calculated and the first address for this subarray is stored in the address table A_{d3}. In Figure 3.3(d) an extension along dimension d_4 is done. As a result of the extension history counter becomes 3, segmented subarray size becomes $l_1 \times l_3 = 1 \times 2 = 2$, and number of segments are $l_2 = 2$; since number of segments depends on the length of adjacent dimension. Therefore, H_{d4} memorizes the history value 3, and A_{d4} has two entries. And $C_{d4}[2]= 2$, since current length of dimension 3, l_3=2. Similarly, the extension along direction d_1 is shown Figure 3.3(e) and finally Figure 3.3(f) shows one more extension along dimension d_3.

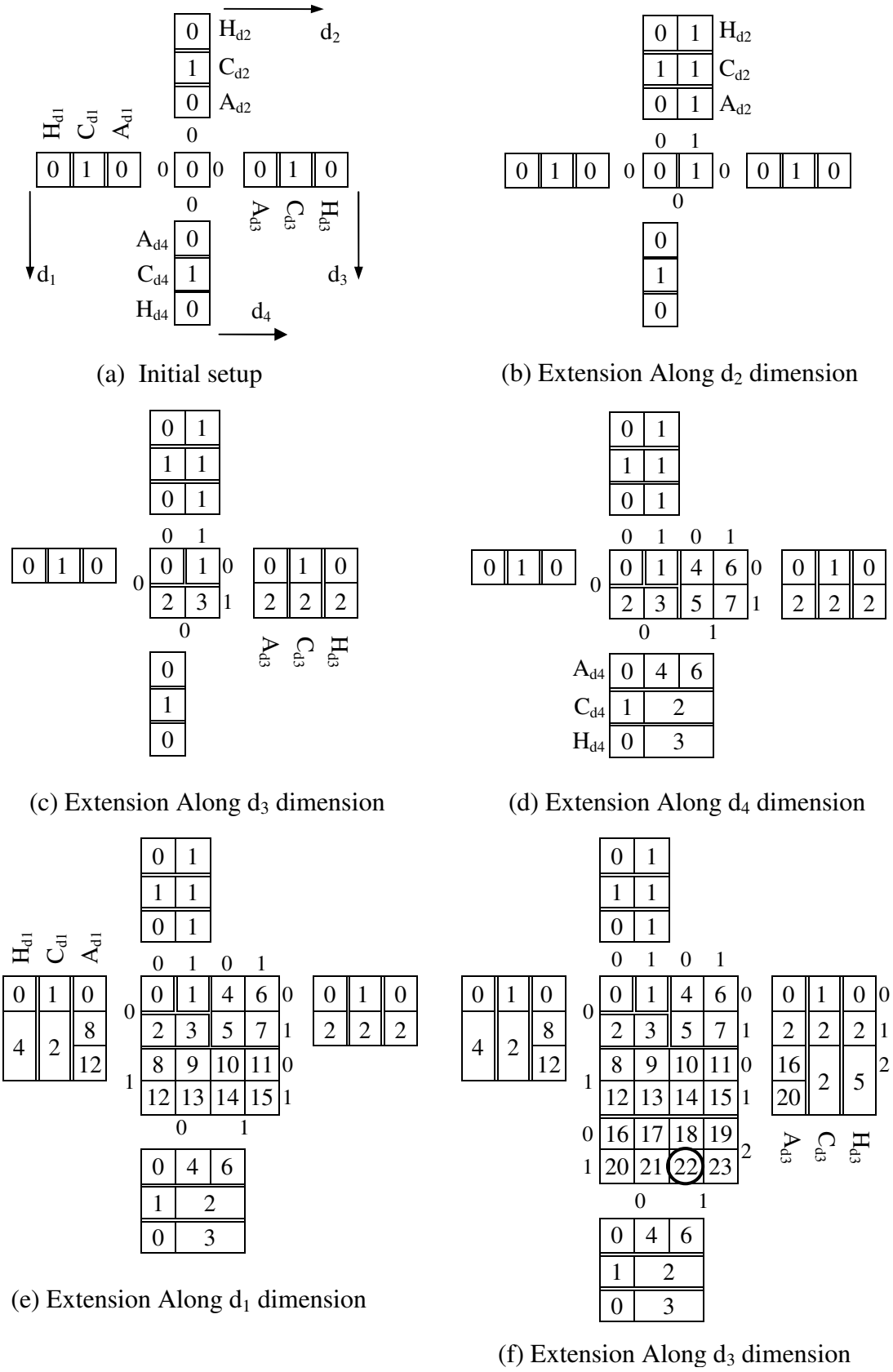
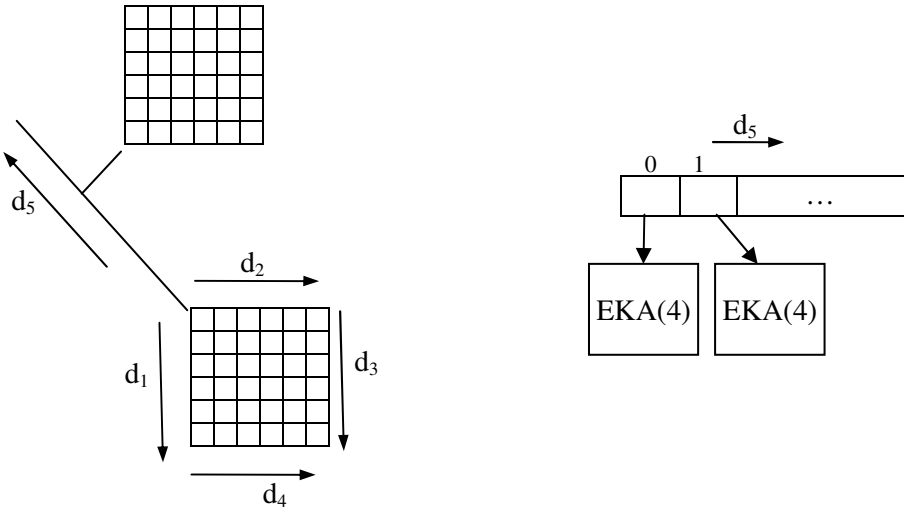


Figure 3.3 : Extension realization of EKA(4).

3.4 Generalization of EKA to Higher Dimensions

The EKA scheme can be generalized to *n* dimensions using a set of EKA(4)s – that is an EKA(*n*) is a collection of EKA(4) which is represented a hierarchical tree like structure. The highest or *n*th dimension will be the root of the tree, the subsequent dimensions up to dimension 5 are the internal node of the tree, and the lowest 4 dimensions presented as EKA(4) and act as leaf in the tree. Figure 3.4(a) shows the logical structure and Figure 3.4(b) shows the physical implementation of a EKA(5) where the length of dimension d_5 is 2. Figure 3.5 shows an EKA(6) represented by a set of EKA(4) in two level. If the current length of dimension d_5, and d_6 is 3 and 2 respectively, then the EKA(6) is represented by the two level structure as shown in Figure 3.5. Each higher dimensions (d_5 and d_6) are represented as one dimensional array of pointers that points to the next lower dimension and each cell of d_5 points to each of the EKA(4). So each EKA(4) can be accessed simply by using the subscripts of higher dimensions. For the case of EKA(n), similar hierarchical structure will be needed. The set of EKA(4)s stores the actual data values and the hierarchical arrays are indexes to EKA(4)s and used to locate the appropriate EKA(4). Hence the EKA(n) is a set of EKA(4)s and a set of pointer is used for indexing purpose only. At this stage (Figure 3.5), if dimension d_1 (or d_2, d_3, d_4) is extended dynamically, all the EKA(4)s will be extended along that dimension and the auxiliary tables are maintained.



(a) Logical view of EKA(5) (b) Physical implementation of EKA(5)

Figure 3.4: Realization of 5-dimensional EKA

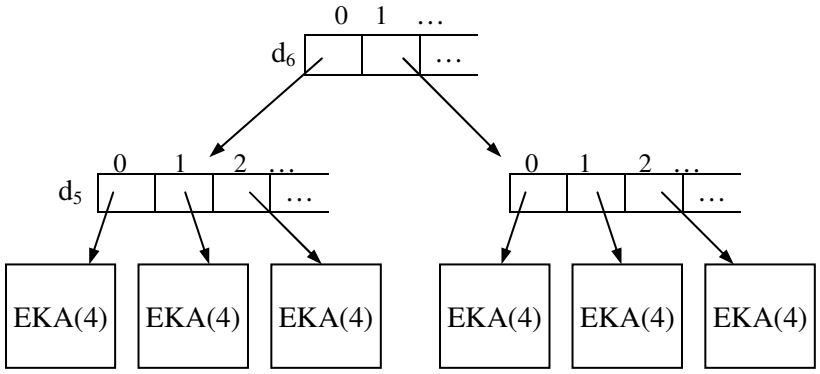


Figure 3.5: Realization of 6-dimensional EKA

3.5 Basic Operations on EKA

The basic operations on any data structure are insertion or update of an array cell, extend the length of any dimension, reduction of length for any dimension, and more importantly retrieval of array element from a particular cell, or from a range of cells [33]. For insertion or update, and retrieval of an element from a cell, some subscript for all dimension are given – and we have to locate the exact array cell to do the required operation. When subscripts for all known dimension are given to locate the cell, this type of query is called point key query. When subscript of only one dimension is given with a single value - the query is called single key query, or given a range of values - the query is called range key query. First we describe how to perform these types of query which is given beneath. Later length extension or reduction is described.

3.5.1 Point Query

In an n-dimensional array, point key query can be defined as – there should be given subscripts for all n dimensions, e.g. $\langle x_1, x_2, ..., x_{n-1}, x_n \rangle$. How this type of query can be performed on EKA is described here in two stage, firstly on EKA(4), then on EKA(n) with $n > 4$.

Point Query on EKA(4)

Let the value to be retrieved is indicated by the subscript $\langle x_1, x_2, x_3, x_4 \rangle$. The maximum history value among the subscripts h_{max}= max($H_{d1}[x_1]$, $H_{d2}[x_2]$, $H_{d3}[x_3]$, $H_{d4}[x_4]$) and the dimension (say d_{max}) that corresponds to history value h_{max} is determined. h_{max} is the

subarray that contains the desired element. This is because; the history values are linear numbers, therefore subarray having maximum history value was constructed at last. Hence the desired element remains in the segment that makes the subarray having history value h_{max}. Now the first address and offset from the first address is to be found out. The adjacent dimension $adj(d_{max}) = d_{adj}$ (say) and its subscript x_{adj} is found. Now the first address of the segment is found from $A_{dmax}[x_{max}][x_{adj}]$. The *offset* from the first address is computed using the addressing function (described in Section 3.1); the coefficient vectors are stored in C_{dmax}. Then adding the *offset* with the first address, the desired array cell (x_1, x_2, x_3, x_4) is found. More concretely we can write it as follows:

h_{max}= max($H_{d1}[x_1]$, $H_{d2}[x_2]$, $H_{d3}[x_3]$, $H_{d4}[x_4]$)

d_{max} = dimension corresponding to h_{max}

x_{max} = given subscript corresponding to d_{max}

$d_{adj} = adj(d_{max})$

x_{adj} = given subscript corresponding to d_{adj}

x_{oth} = given subscripts of dimensions other than x_{max} and x_{adj}

$firstAddress = A_{dmax}[x_{max}][x_{adj}]$

$offset = C_{dmax}[x_{max}]*x_{oth1} + x_{oth2}$

$cellPos = firstAddress + offset$

Example 3.1: Let four subscripts ⟨1, 0, 2, 1⟩ for dimension d_1, d_2, d_3, and d_4 is given (see Figure 3.3(f)). Here h_{max} = max($H_{d1}[1]$, $H_{d2}[0]$, $H_{d3}[2]$, $H_{d4}[1]$)= max(4, 0, 5, 3) =5, and dimension corresponding to h_{max} i.e. $d_{max} = d_3$ whose subscript $x_{max} = 2$ and $adj(d_{max}) = adj(d_3) = d_1 = d_{adj}$ and $x_{adj} = 1$. So the $firstAddress = A_{d3}[2][1] = 20$, and offset is calculated using the coefficient vector stored in coefficient table C_{d3} which is 2. Here, $offset = C_{d3}[2]*x_4 + x_2 = 2*1 + 0 = 2$. Finally adding the offset with the first address the desired location 20 + 2 = 22 is found (encircled in Figure 3.3(f)).

Point Query on EKA(n), $n > 4$

Let the value to be retrieved is indicated by the subscript ⟨x_n, x_{n-1}, …, x_2, x_1⟩. Each of the higher dimensions ($n > 4$) is the set of one dimensional pointer arrays that points to next lower dimensions (see Figure 3.5). Hence using subscripts x_k ($k > 4$) the pointer arrays are

searched to locate the appropriate EKA(4). Then using the computation technique described in section 3.6.1 for lower four subscripts the exact cell location in EKA(4) can be found.

3.5.2 Range Query

A range key query [34,35] has a single predicate of the form (*column subscript* < *value*) or (*column subscript* > *value*) or (*column subscript between value$_1$ and value$_2$*). On the other hand, for a single key query predicate has the form *column subscript* = *value.* So we can say that single key query is a special case of range key query with only a single range subscript.

Range Query on EKA(4)

Let the specified range involve in the known column has subscripts x_{k1}, x_{k2}, …, x_{kNRQ} of dimension d_k (k = 1, 2, 3, 4). Let h_1, h_2, … h_{NRQ} be the history values that correspond to the subscripts and the minimum history value be $h_{min} = \min(h_1, h_2, \ldots h_{NRQ})$.

Definition 3.2 (Major and Minor subarray) All the elements of the subarrays corresponding to the history values h_1, h_2, … h_{NRQ} are called *Major subarray*. The subarrays that have history values greater than h_{min} and belong to the adjacent dimension *adj*(d_k) are called *Minor subarray*. The candidate subarrays are those which are sufficient to be searched and these subarrays have history values greater than or equal to h_{min}.

For Range key query all the major subarrays and one or more segments of the minor subarrays are candidate subarrays. The subarrays do not belong to the known dimension d_k or *adj*(d_k) and have history values greater than h_{min} are also candidate subarray, but from here the desired element is extracted by point key query.

For single key query only one major subarray and exactly one segment of minor subarrays are candidate subarrays and the rest are same as range key query.

Example 3.2: Figure 3.6, which is obtained by extending the Figure 3.3(f) in d_2 dimension, shows the candidate range (bold dotted line) of a range key query for a EKA(4). Assume that the candidate range of the subscripts of the corresponding dimension d_1 has *NRQ* subscripts from 1 to 2.

In Figure 3.6, since known subscripts are $x_{11} = 1$ and $x_{12} = 2$ of dimension d_1, i.e. the query is ⟨1-2, *, *, *⟩, the subarray having history values 4 and 7 (as $H_{d1}[1] = 4$, and $H_{d1}[2] = 7$)

are the major subarray. On the other hand, the subarray having history values 5 is the minor subarray. Hence all the elements of subarray 4 and 7 are candidate for retrieval and one segment of subarray 5 are candidate for retrieval. Here, subarray 6 is the only remaining candidate subarray, since it has history value greater than 4 (definition 3.2) and the elements inside the subarray are found by calculating the offsets and adding the first address as described in section 3.5.1

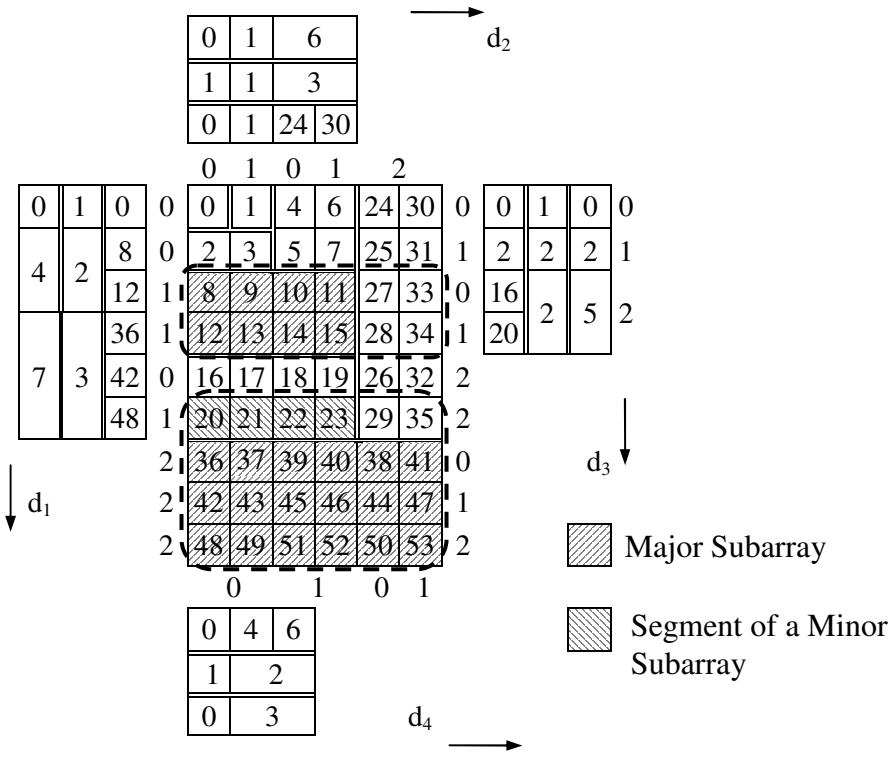


Figure 3.6: Range query on EKA(4).

Range Query on EKA(*n*), *n* > 4

Let the specified range involve in the known dimension has subscripts $x_{k1}, x_{k2}, \ldots, x_{kNRQ}$ of dimension d_k ($k > 4$). In this case the, using the other dimensional subscripts x_j ($j > 4$) ($x_j = 0$ to $l_j - 1$) along with known dimension NRQ subscripts simply search the higher dimensional index pointers arrays to find the appropriate EKA(4). In this case, all the elements of the searched EKA(4) are candidate for retrieval, so simply retrieve them.

Again if known subscripts $x_{k1}, x_{k2}, \ldots, x_{kNRQ}$ of dimension d_k ($k = 1, 2, 3, 4$), then all the EKA(4)s are candidate to be accessed. So iteratively apply the technique described above for that NRQ subscripts to each of the EKA(4).

Example 3.3: Let we have NRQ subscripts 1 – 2 of dimension d_5, then all the elements of the desired EKA to be retrieved are shown in rectangle in Figure 3.7. Again, if those NRQ subscripts were for dimension d_3, elements to be retrieved from all the EKA(4)s shown in Figure 3.7 as left-upward shading.

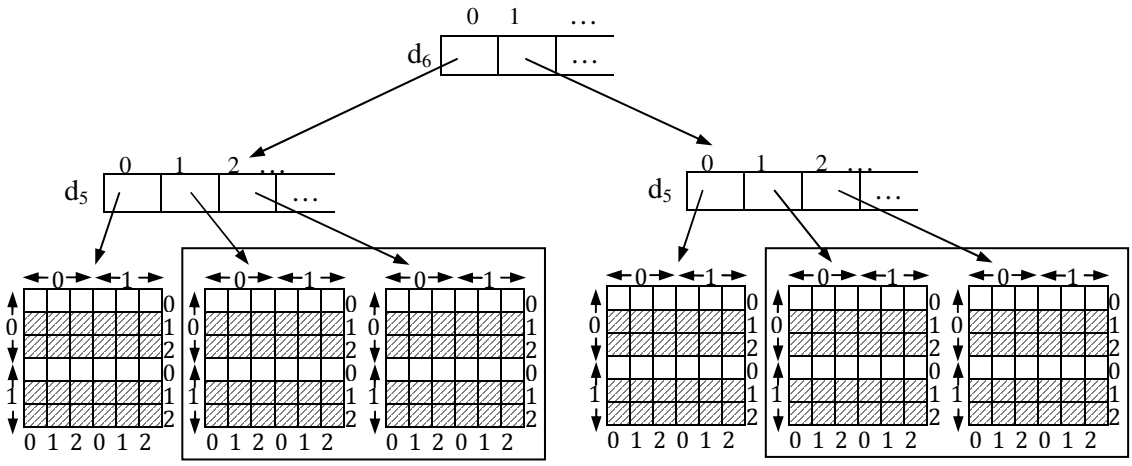


Figure 3.7: Range query on EKA(6).

3.5.3 Increment Operation

The increment operation can be defined as extending the size of the array by extending any length of arbitrary dimension or introducing a new dimension. We call this operation as extension. The presented EKA is an extendible array, where the length of each dimension can be extended to any length with the condition that extension is made on the boundary of that dimension. That is there is no facility to insert a subarray midst of any dimension. This seems problematic, but most of the real world applications need incremental extension only. So we allowed dynamic extension only at the boundary of each dimension in proposed EKA. The detail of extension process, how the auxiliary tables are maintained are explained is section 3.4 for EKA(4) and in section 3.5 for EKA(*n*).

3.5.4 Reduction Operation

The reduction of size of the array EKA is possible with the prerequisite that the deletion or reduction of length is made only at the perimeter of the array. Moreover, it is not possible to reduce the length of any arbitrary dimension, whereas we can only reduce the most recently extended dimension. That is for deletion one has to go through the reverse way of

how the extension is made. For this purpose we can maintain a stack which is populated during each extension and points to the extended subarray. So when deletion is necessary, simply pop the link from stack, free the storage and update necessary parameters in auxiliary table.

3.6 Theoretical Analysis

In this section, we model the processes of retrievals and extensions for multidimensional array under two different implementation strategies namely Traditional Multidimensional Arrays (TMA) and our proposed Extendible Karnaugh Arrays (EKA). The TMA reorganizes the array whenever there is an extension to it. That is, the whole array will be relinearized on disk to accommodate the new data due to the extension of length of dimension. Here, we show that the EKA strategy can reduce the cost of array extensions significantly. We will derive the cost functions for both extensions and retrievals in the following. All the array schemes are assumed to be stored in secondary storage and performed the operations.

3.6.1 Parameters

The cost functions are represented as the number of array cells required to access. The parameters that are assumed are described in Table 3.1. All the lengths are in bytes. Some parameters are provided as input while others are derived from input parameters.

Table 3.1: Parameters for cost function for TMA and EKA

Parameter	Description
n	Number of dimension both for TMA and EKA
EKA(n)	An n dimensional Extendible Karnaugh Array
TMA(n)	An n dimensional Traditional Multidimensional Array
d_i	Dimension i, $1 \leq i \leq n$
l_i	Length of dimension d_i
V	Initial volume of both TMA and EKA, $V = \prod_{i=1}^{n} l_i$
s	Number of segments in a subarray for EKA
λ	Length of extension
SE_{di}	Size of extension along dimension d_i

$EC_{\lambda}^{EKA(n)}$	Extension cost of n dimensional EKA with extension length λ in each dimension
$EC_{\lambda}^{TMA(n)}$	Extension cost of n dimensional TMA with extension length λ in each dimension
FC_{TMA}	Read (Face) cost of TMA
RC_{TMA}	Relocation cost of TMA
$EG_{n,\lambda}$	Extension gain of EKA(n) over TMA(n) for λ extension in each dimension

Assumptions:

To simplify the cost model we make a number of assumptions.

(i) The length of dimensions extends in round robin manner of the dimensions for both TMA and EKA.

(ii) The length of each dimension is equal and when extension occurs each of the dimensions are extended by equal length. We denote the length of dimension after *i*th extension as l_i.

(iii) All the basic CPU operations are executed in constant time.

3.6.2 Retrieval Cost

In TMA, the array is linearized in a single data stream using the addressing function described in section 2.2 and all offset values of the array elements are consecutive. Hence the range of candidate offset values for a query can be determined uniquely. But for EKA, the same data stream is distributed over different subarrays (See Figure 3.6).

Cost for TMA

The retrieval on TMA is dependent on the known dimension (i.e. the specified dimension) of query dimension. We use the term *known* dimension (or known subscript) to indicate the specified dimension of the query operation. For example dimension 2 is the known or specified (i.e. subscript x_2 *is known*) dimension in Figure 3.6.

In an *n* dimensional TMA, if the query is along dimension *n* (i.e. subscript i_n is known) then all the candidate offsets are consecutive and the volume of the range of the query is l_i^{n-1}. This is explained with an example in the following. For a 4-dimensional array with

length of each dimension $l_i = l$ the addressing function can be written from equation 2.1 as follows.

$$f(x_4, x_3, x_2, x_1) = l^3x_4 + l^2x_3 + lx_2 + x_1$$

If l=6 and x_4 is known (say, $x_4 = 0$, and x_j =0,…,l-1 for j=1,…,3) then the candidate offset values in the query are consecutive in the range 0 to 215 (total 216 offsets) out of 1296 offsets which is l^3 (i.e. 6^3). If x_1 is known (say, x_1=0) then the candidate offset values in the query are in the range 0 to 1290 (total 1291 offsets) out of 1296 offsets. Hence the volume of the candidate range of target elements are determined by $l^4 - (l - 1)$. If the subscript x_2 is known then the volume of the candidate range of offsets is $l^4 - l(l - 1)$. In general, if the subscript x_k ($1 \le k \le n$) is known then the volume of the target elements are determined by $l^n - l^{k-1}(l - 1)$. For the range key query in the range of known subscripts NRQ along the dimension k, the volume of the target elements are determined by $NRQ \times (l^n - l^{k-1}(l - 1))$.

From the above discussion, we can conclude that the retrieval in TMA is largely depends on the known dimension k and when k = n then the retrieval time will be minimum and when k = 1 then the retrieval time will be maximum.

Example: Consider a 3D array of size 3×3×4 stored as row major order shown in Figure 3.8(a). If we consider the known dimension is 1, and the known subscript $x_1 = 0$ then, the candidate values to be retrieved are shown in Figure 3.8(b), or if $x_1 = 1$ then, the candidate values can be as in Figure 3.8(c), and so on. So for a retrieval considering the first dimension as known dimension, each of the candidate values are totally discrete and spread over the entire range. If the known dimension is 2, and $x_2 = 0$ or 1, then the candidate values can be in Figure 3.8(d) and 3.8(e) respectively, and so on. Here some of the candidate values are grouped together, though it can cover the entire range. If the known dimension is 3, and $x_3 = 0$, then the candidate values can be in Figure 3.8 (f), and so on. Here many of the candidate values are contiguous in nature and needs only single read since it is the highest dimension.

Cost for EKA

In EKA scheme, the target elements are distributed in different subarrays which are further divided into segments. So for retrieval operations, EKA will take more CPU operation to be performed for accessing different streams in secondary memory. But on the other hand

each of the segments of the subarray is two dimensional and candidate and non candidate items can be separated in EKA. And thus retrieval cost will be lower. As the segments are 2 dimensional then the maximum volume of the target elements for a query in a segment is determined by $NRQ \times (l^2 - (l-1))$. If the number of segment is s then the maximum volume of the target elements are determined by $s \times NRQ \times (l^n - l^{k-1}(l-1))$, where s depends on the size of the subarray.

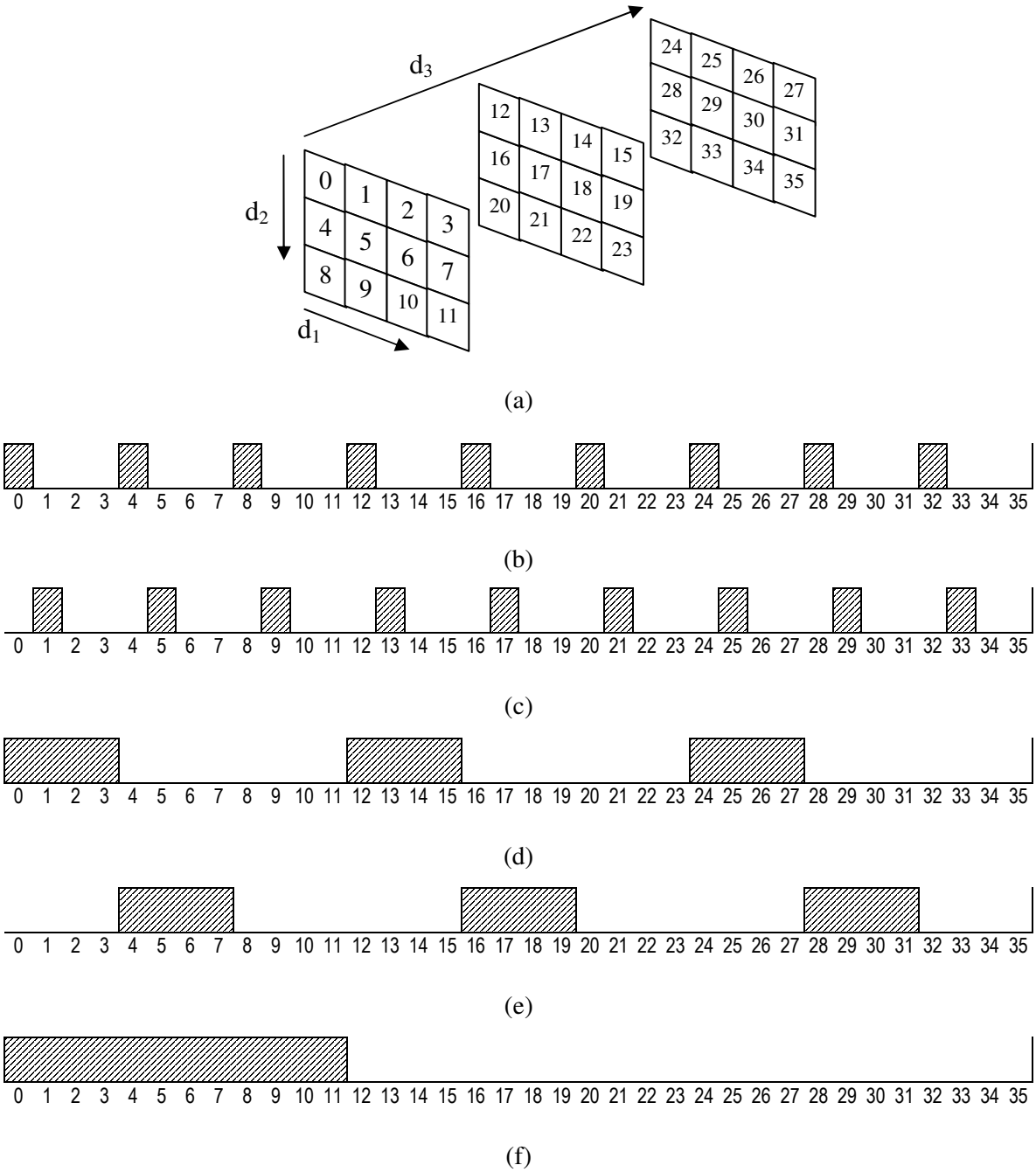


Figure 3.8: A 3-dimensional TMA and its retrieval candidates.

3.6.3 Extension Cost

Cost for EKA

Figure 3.9 shows the pictorial view of λ unit extension of EKA(4), EKA(5), and EKA(6). By λ unit extension we mean that all dimensions of each EKA are extended a value λ. Each pair in Figure 3.9 shows the before and after view of extension.

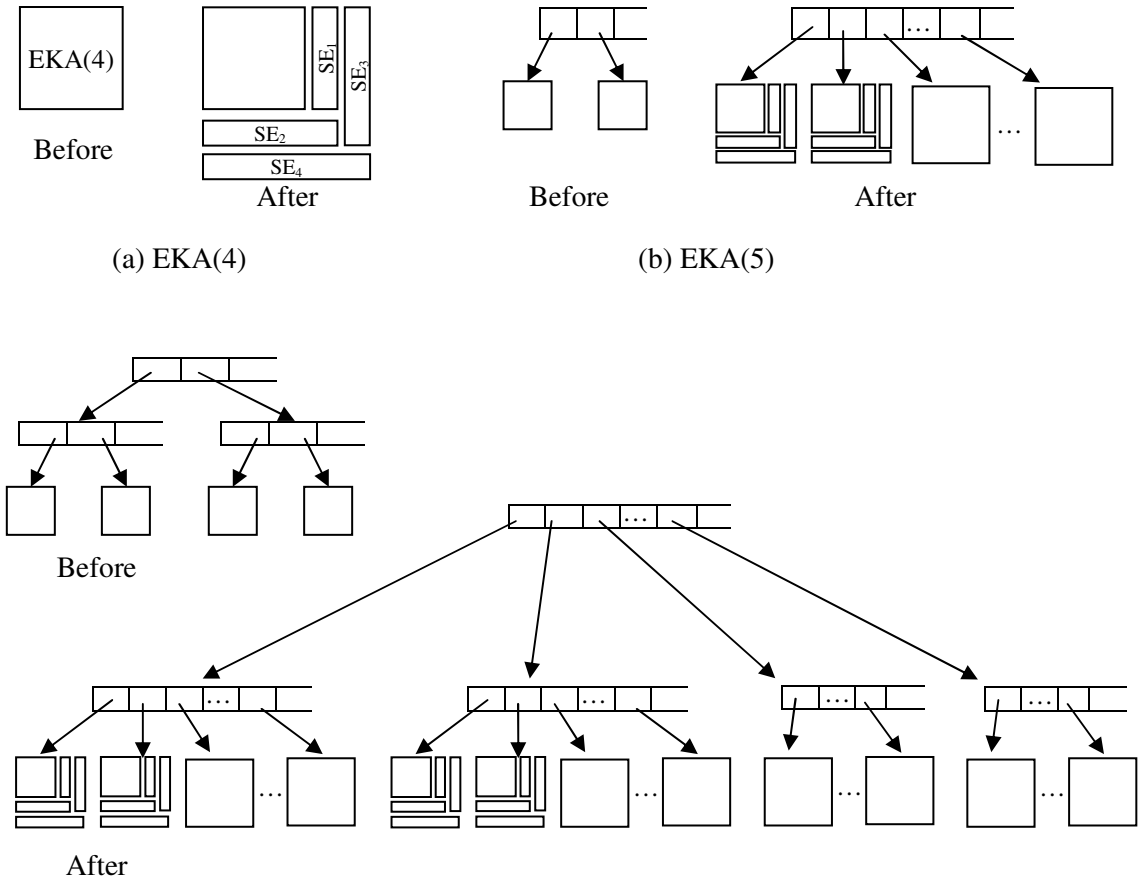


Figure 3.9: Extension cost analysis of EKA.

Let us consider EKA(4) with length of each dimension $l_i = l$. So, initial volume of the array before extension $V = l_1 \times l_2 \times l_3 \times l_4 = l^4$

If we extend one unit along each dimension d_i, the size of extension SE_i is

$SE_1 = l_2 \times l_3 \times l_4 = l^3$, and due to extension $l_1 = l + 1$

$SE_2 = l_1 \times l_3 \times l_4 = (l + 1)l^2$, and due to extension $l_2 = l + 1$

$SE_3 = l_1 \times l_2 \times l_4 = (l + 1)^2 l$, and due to extension $l_3 = l + 1$

$SE_4 = l_1 \times l_2 \times l_3 = (l + 1)^3$, and due to extension $l_4 = l + 1$

Now in general, extending a λ unit along dimension d_i, the size of extension SE_i can be written as

$SE_1 = \lambda \times l_2 \times l_3 \times l_4 = \lambda l^3$, and due to extension $l_1 = l + \lambda$

$SE_2 = \lambda \times l_1 \times l_3 \times l_4 = \lambda(l + \lambda) l^2$, and due to extension $l_2 = l + \lambda$

$SE_3 = \lambda \times l_1 \times l_2 \times l_4 = \lambda(l + \lambda)^2 l$, and due to extension $l_3 = l + \lambda$

$SE_4 = \lambda \times l_1 \times l_2 \times l_3 = \lambda(l + \lambda)^3$, and due to extension $l_4 = l + \lambda$

So, Total Extension Cost for EKA(4), λ unit extension in each dimension, becomes

$$EC_{\lambda}^{EKA(4)} = SE_1 + SE_2 + SE_3 + SE_4$$

$$= \lambda \sum_{i=0}^{k} l^{k-i} (l + \lambda)^i \text{ ,where } k = 3$$

Now Consider EKA(5), with initial volume of the array before extension V = l^5 (considering length of each dimension $l_i = l$)

Extending a λ unit along dimension d_i, the size of extension SE_i is

$SE_1 = \lambda \times l_2 \times l_3 \times l_4 \times l_5 = \lambda l^4$, and due to extension $l_1 = l + \lambda$

$SE_2 = \lambda \times l_1 \times l_3 \times l_4 \times l_5 = \lambda(l + \lambda) l^3$, and due to extension $l_2 = l + \lambda$

$SE_3 = \lambda \times l_1 \times l_2 \times l_4 \times l_5 = \lambda(l + \lambda)^2 l^2$, and due to extension $l_3 = l + \lambda$

$SE_4 = \lambda \times l_1 \times l_2 \times l_3 \times l_5 = \lambda(l + \lambda)^3 l$, and due to extension $l_4 = l + \lambda$

$SE_5 = \lambda \times l_1 \times l_2 \times l_3 \times l_4 = \lambda(l + \lambda)^4$, and due to extension $l_5 = l + \lambda$

Total Extension Cost for EKA(5), λ unit extension in each dimension, becomes

$$EC_{\lambda}^{EKA(5)} = SE_1 + SE_2 + SE_3 + SE_4 + SE_5$$

$$= \lambda \sum_{i=0}^{k} l^{k-i} (l + \lambda)^i \text{ ,where } k = 4$$

Similarly for EKA(n) Total Extension Cost, for λ unit extension in each dimension, can be written as

$$EC_{\lambda}^{EKA(n)} = SE_1 + SE_2 + SE_3 + \ldots + SE_{n-1} + SE_n$$

$$= \lambda \sum_{i=0}^{k} l^{k-i} (l + \lambda)^i \text{ ,where } k = n-1 \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (3.1)$$

If we expand the summation, then

$$\sum_{i=0}^{k} l^{k-i}(l+\lambda)^i = l^k(l+\lambda)^0 + l^{k-1}(l+\lambda)^1 + l^{k-2}(l+\lambda)^2 + ... + l^1(l+\lambda)^{k-1} + l^0(l+\lambda)^k$$

$$\begin{aligned} &= l^k + \\ &\quad l^{k-1}({}^1C_0 l + {}^1C_1\lambda) + \\ &\quad l^{k-2}({}^2C_0 l^2 + {}^2C_1\lambda l + {}^2C_2\lambda^2) + \\ &\quad l^{k-3}({}^3C_0 l^3 + {}^3C_1\lambda l^2 + {}^3C_2\lambda^2 l + {}^3C_3\lambda^3) + \\ &\quad l^{k-4}({}^4C_0 l^4 + {}^4C_1\lambda l^3 + {}^4C_2\lambda^2 l^2 + {}^4C_3\lambda^3 l + {}^4C_4\lambda^4) + \\ &\quad : \\ &\quad + \\ &\quad l^0({}^kC_0 l^k + {}^kC_1\lambda l^{k-1} + {}^kC_2\lambda^2 l^{k-2} + ...+ {}^kC_{k-1}\lambda^{k-1} l + {}^kC_k\lambda^k) \end{aligned}$$

After multiplying and collecting the coefficients of l^p, p = 0, 1, …, k, we get

$$\sum_{i=0}^{k} l^{k-i}(l+\lambda)^i = l^k \sum_{i=0}^{k} {}^iC_0 + l^{k-1}\lambda \sum_{i=1}^{k} {}^iC_1 + l^{k-2}\lambda^2 \sum_{i=2}^{k} {}^iC_2 + ... + l\lambda^{k-1} \sum_{i=k-1}^{k} {}^iC_{k-1} + \lambda^k \sum_{i=k}^{k} {}^iC_k$$

$$= {}^{k+1}C_1 l^k + {}^{k+1}C_2 l^{k-1}\lambda + {}^{k+1}C_3 l^{k-2}\lambda^2 + ...+ {}^{k+1}C_k l\lambda^{k-1} + {}^{k+1}C_{k+1}\lambda^k$$

$$\left[\text{Since } \sum_{j=0}^{p} {}^jC_r = {}^{p+1}C_{r+1}\right]$$

$$= \sum_{i=1}^{n} {}^nC_i l^{n-i}\lambda^{i-1}, \ \textit{where } n = k+1$$

Putting the above value in equation (3.1), we get

$$EC_{\lambda}^{EKA(n)} = \lambda \sum_{i=0}^{k} l^{k-i}(l+\lambda)^i \text{ ,where } k = n-1$$

$$= \lambda \sum_{i=1}^{n} {}^nC_i l^{n-i}\lambda^{i-1}$$

$$= \sum_{i=1}^{n} {}^nC_i l^{n-i}\lambda^i \qquad \text{..... (3.2)}$$

Cost for TMA

Consider the Figure 3.10(a), which shows a 2D TMA of size 3×4. Let its cell values represent the location of each cell after linearization. From Figure 3.10(a) we find that the location of cell ⟨1,2⟩ is 6. Now let we want to extend the array one unit in d_2. Figure 3.10(b) shows the array after extension, from where we see that location of cell ⟨1,2⟩ is now 7.

That is if we simply append the extension subarray at the end, we will get wrong value of cell ⟨1,2⟩. To get the correct one we first need to read the previously allocated data and then reorganize the array.

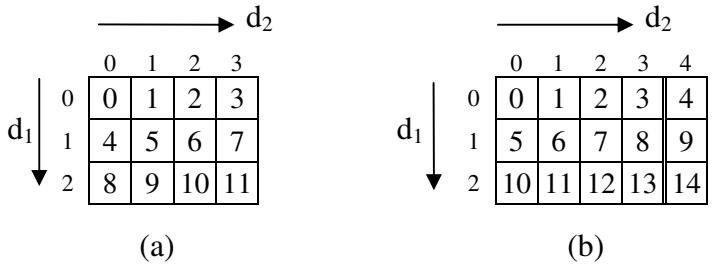


Figure 3.10: A 2D TMA and its extension.

Let us now consider a TMA(n), with each dimension length $l_i = l$

So initial volume V = $l_1 \times l_2 \times l_3 \times \ldots \times l_n = l^n$

We already seen that for extending TMA, it requires to reorganize the array and rewrite both existing and new data elements. The existing elements of the initial array need to be faced and recalculate the new offsets due to the extension for TMA.

Hence the cost of facing (FC) the existing array elements becomes

$FC_{TMA} = V = l^n$

If a TMA is extended by λ then a new TMA of length $l + \lambda$ is to be reallocated, Hence reallocation cost $RC_{TMA} = (l_1 + \lambda) \times (l_2 + \lambda) \times (l_3 + \lambda) \times \ldots \times (l_n + \lambda) = (l + \lambda)^n$

So, Total extension cost for TMA(n)

$$EC_{\lambda}^{TMA(n)} = FC_{TMA} + RC_{TMA}$$

$$= l^n + (l + \lambda)^n = l^n + \sum_{i=0}^{n} {}^nC_i l^{n-i} \lambda^i = l^n + {}^nC_0 l^n + \sum_{i=1}^{n} {}^nC_i l^{n-i} \lambda^i$$

$$= 2l^n + \sum_{i=1}^{n} {}^nC_i l^{n-i} \lambda^i \qquad \ldots\ldots\ldots\ldots\ldots (3.3)$$

Extension Gain

The difference of extension cost between the TMA and EKA schemes is referred to as *Extension Gain* (EG)

$EG_{n,\lambda} = EC_{\lambda}^{TMA(n)} - EC_{\lambda}^{EKA(n)}$ = eq. (3.3) − eq. (3.2) $= 2l^n = 2V$.

That is the extension gain is constant (twice of the initial volume) for any values of λ with a fixed initial volume. But it is worth mentioning that this gain is in theoretical aspect. Practically, EG would be little less, because there will some cost increase due to populating those auxiliary tables we have used.

3.6.4 Overflow Cost

In multidimensional array, the location of an element is calculated using the addressing function described in Section 2.2.1. For an n dimensional array with each dimension length = l, maximum value of the coefficient vector can be l^{n-1} which is again multiplied by subscript value (maximum l-1). So the resulted value can be written approximately as l^n. This value quickly reaches the machine limit for TMA (e.g. for 32 bit machine maximum value can be 2^{32}) and thus overflows. But in EKA since each of the segments are two dimensional, this maximum value will be l^2, which greatly delays the overflow.

For example, theoretically for a 32bit address space with TMA(4) the maximum length of each dimension can be 256 but for EKA(4) it can be 65536 which is far greater then TMA's length. For TMA(5) and TMA(6) this maximum length will be much less, but for EKA it remains same. The exact calculation is shown below considering that the length of each dimension is l.

$l^4 = 2^{32}$	$l^2 = 2^{32}$
$\Rightarrow 4\log_2 l = \log_2(2^{32}) = 32$	$\Rightarrow 2\log_2 l = \log_2(2^{32}) = 32$
$\Rightarrow \log_2 l = 8$	$\Rightarrow \log_2 l = 16$
$\Rightarrow l = 2^8 = 256$	$\Rightarrow l = 65536$
For TMA(4)	For EKA

One more practical reason is that TMA requires consecutive memory locations up to l^n during implementation and hence it overflows soon when l and n is large. On the other hand, in EKA the segments of the subarrays are always two dimensional and distributed. Hence consecutive memory location requirement is less in EKA than TMA. Therefore EKA delays the overflow situation even for large values of l.

3.7 Conclusion

In this chapter we present our proposed model in detail, that is, how the model can be realized or implemented with the facility of dynamic extension but excluding the already

stored data reorganization. We present a concept of segment to limit the address space overflow. The EKA doesn't prevent overflow to occur, rather it holds up the occurrence of address space overflow. How the most basic array operations can be performed on EKA are also manifested here. In the next chapter we are going to present a compression scheme which is based over EKA.

CHAPTER IV

A Compression Scheme Based on Extendible Karnaugh Array

4.1 Introduction

Many statistical applications use set of multidimensional arrays to allow the efficient and convenient storage and retrieval of large volumes of data that is closely related, viewed and analyzed from different perspectives. For these applications, data compression is important because performance strongly depends on the amount of available memory. The most obvious outcome of data compression is that it reduces storage cost by storing more logical data per unit of physical capacity [36,37]. Performance is improved because there is less physical data to retrieve during scan-oriented queries. Performance is further enhanced since data remains compressed in memory. More importantly, however, the application of data compression in reducing the cost of data communication in distributed networks. In some other applications like some index structures, it is possible through compression to pack more keys into each index block [38,39,40]. When the database is searched for a given key value, the key is first compressed and the search over the index blocks. The ultimate effect is that fewer blocks have to be retrieved and thus the average search cost is improved.

The compression techniques usually provide two mappings [32,41]. One is forward mapping, computing the location in the compressed dataset given a position in the original dataset. The other one is backward mapping, computing the position in the original dataset given a location in the compressed dataset. A compression method is called mapping-complete if it provides forward mapping and backward mapping. The term logical database and physical database is used to refer to the uncompressed and compressed database respectively. The multidimensional arrays that are linearized to store multidimensional datasets normally have high degree of sparsity and need to be compressed [42,43]. It is therefore desirable to develop techniques that can access the data in their compressed form and can perform logical operations directly on the compressed data. In Chapter II we have already presented some existing and well understood data

compression methods, most of which are not suitable for extendible array. Here we are presenting a compression scheme suitable for our proposed EKA.

4.2 The History Segment-Offset Compression

The History Offset Compression scheme is essentially suitable for Traditional Extendible Array[6,44,45]. The basic scheme is presented in chapter II. Though EKA has different logical structure from Traditional Extendible Array, History Offset Compression can easily be incorporated over EKA with some modification. Since EKA has already a history counter that points an extended subarray which is further divided into segments, if the segments are sparse we can store the value as well as the offset from the starting of the segment in the physical array. As because the offsets are within a segment, we call the scheme as History Segment-Offset Compression (HSOC).

4.2.1 Realization of HSOC on EKA(4)

In addition to the auxiliary tables of EKA mentioned in chapter III, there is also an auxiliary array named *Element* needed for all dimension to store the number of elements in the segment. Though there may be several segments in an extended subarray, only one entry in *Element* array for each subarray is sufficient to retrieve the value accurately. *Element* will store the number of elements in the last segment or the one and only segment of the extended subarray. All these auxiliary tables are sufficient for EKA(4) to be mapping complete, but for higher dimensional EKA we need some other auxiliary tables that is explained in next section.

Consider the following logical structure of EKA(4) in Figure 4.1(a) which is actually the real array of Figure 3.6. Here the cell values represent the value as well as the offset of that cell in physical array. Now let us consider that only shaded squares represent that there is a valid value on the cell and other cells are empty. The history offset compressed representation of the array is shown in Figure 4.1 (b). Here, the *History tables*, and *Coefficient tables* are as before, *Address table* points to the starting physical address of the segment if there is some elements in the segment otherwise it is null. *Element table* maintain the number of elements in a subarray. For example $E_{d1}[2] = 3$, because subarray 7 has three segments, one of which is empty and the last segment has 3 elements. In the centre of Figure 4.1(c), the physical array is placed. Here we will see that, each of the non-

empty array value is placed along with its offset - i.e. displacement of that value in the segment. For example array value 13, 14 have offset 1, 2 respectively which are stored in the physical array. Here, the values are stored in sorted fashion according to their offsets for efficient retrieval.

Forward Mapping on Compressed EKA(4)

Let the value to be retrieved is indicated by the subscript $\langle x_1, x_2, x_3, x_4 \rangle$. We have to calculate h_{max}, *offset,* and *firstAddress* in similar way described in section 4.3.1. Now if the *firstAddress* is null, the element doesn't exist at all. Otherwise determine the number of elements in the segment, which may be found in *Element* table if it is the only segment or the last segment, else number can be calculated from the difference of *firstAddress*es of the current and next available segment. If each of the array cells consumes *k* bytes in memory or disk, then for exact calculation of number of elements, we have to divide the difference by *k*. And then load the segment from disk to memory and do a binary search to find the offset. If offset is found the corresponding value is the desired one, otherwise there is no such value for those subscripts.

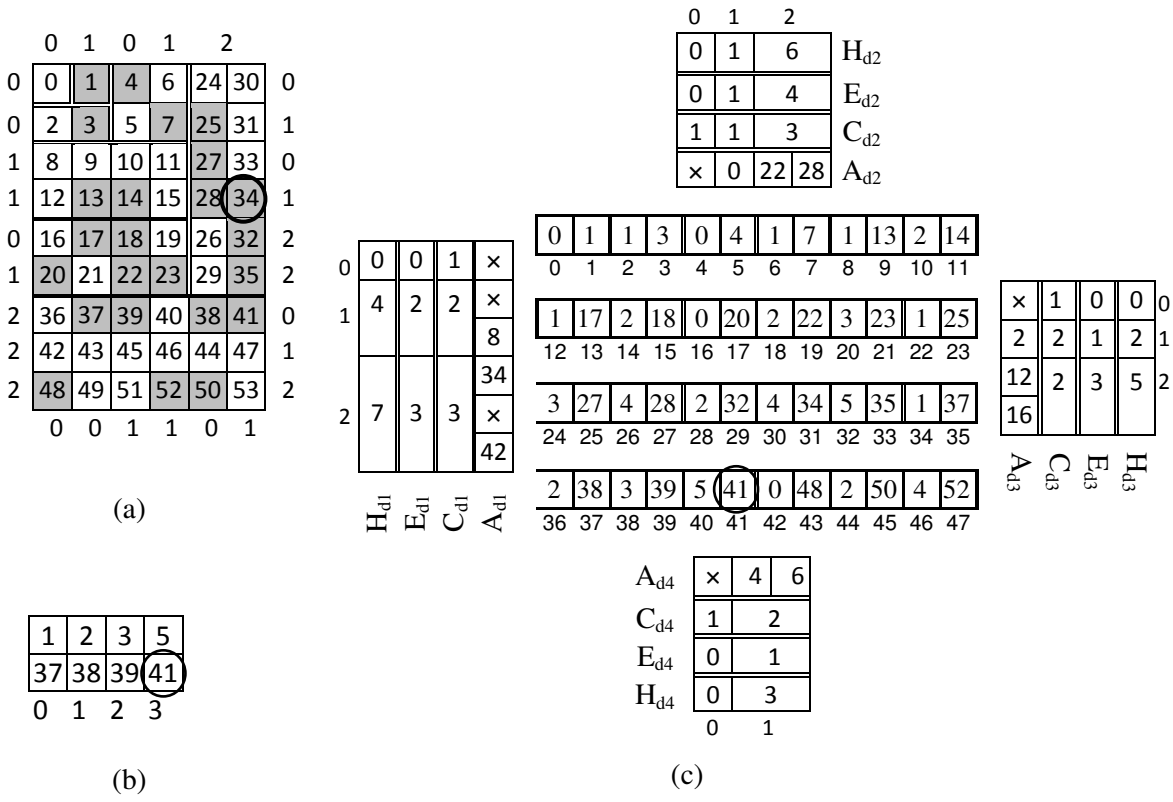


Figure 4.1: History Segment-Offset representation of EKA(4).

Example: Let four subscripts ⟨2, 2, 0, 1⟩ for dimension d_1, d_2, d_3, and d_4 is given (see Figure 5.1). Here h_{max} = max($H_{d1}[2]$, $H_{d2}[2]$, $H_{d3}[0]$, $H_{d4}[1]$)= max(7, 6, 0, 3) =7, and dimension corresponding to h_{max} ie. $d_{max} = d_1$ whose subscript $x_{max} = 2$ and *adj*(d_{max}) = *adj*(d_1) = $d_3 = d_{adj}$ and $x_{adj} = 0$. So the *firstAddress* = $A_{d1}[2][0] = 36$, and offset is calculated using the coefficient vector stored in coefficient table C_{d1} which is 3. Here, *offset* = $C_{d1}[2] * x_4 + x_2 = 3*1 + 2 = 5$. Now the segment is loaded into memory (Figure 4.1(b)), and binary search finds the offset 5, therefore the desired value is 41 (encircled in Figure. 4.1 (b), (c)).

Backward Mapping on Compressed EKA(4)

Let we are given ⟨*h*, *s*, *o*⟩ that represents history value, the segment number, and an offset position respectively in a Compressed EKA(4). We have to determine the subscripts of each dimension. The history values are monotonically increasing and placed sequentially in history table, so we can apply binary search to each of the history table to find the given *h*. Let we found the value in history table of dimension *i* (H_{di}) at position *x*, then subscript of dimension *i* is x_i. Let *adj*(d_i) = d_j, then x_j equals to the provided segment number *s*. Let the coefficient table entry in dimension *i* at *x* is *c* ie. $C_{di}[x]$ = c, then two other dimensional subscripts x_u, x_v (say) can be found by following formula

x_u = offset % c

x_v = offset \ c

where % is a modulus or remainder operator, and \ is a integer division operator.

Example: let the given values are ⟨6, 1, 4⟩ that is history = 6, segment number = 1, offset = 4. Now applying binary search on each history table, we found that $H_{d2}[2] = 6$, so x_2 = 2. Here *adj*(d_2) = d_4, so $x_4 = 1$ (the segment number). Again, we see that $C_{d2}[2] = 3 = c$, which was the length of dimension 3 during extension. So x_3 = offset % 3 = 4 % 3 = 1, and x_1 = offset \ 3 = 4 \ 3 = 1. Hence the subscripts are ⟨1, 2, 1, 1⟩ (encircled in Figure. 4.1 (a)).

4.2.2 Realization of HSOC on EKA(*n*)

We only compress each EKA(4) and upper pointer arrays remains as usual. Since an n-dimensional EKA is collection of EKA(4)s, so we can individually apply the HSOC over each EKA(4)s on a iterative manner. Forward mapping described above (section 4.2.1) can be applied on each of those EKA(4) after reaching there by using the higher dimensional

pointer arrays. But for backward mapping we need some additional tables, since the EKA scheme loses the higher dimensional subscripts. So, each EKA(4) and higher dimensional pointer arrays will maintain a tiny (length = 2) *Uppersubscripts* array. It will contain the index of immediate next higher dimension and a pointer back to that higher n–4 dimensions pointer array. Again, each EKA(4) have their own history tables, so to find the desired EKA(4) where the given history value lies we need to apply binary search all of them. For an EKA(n) with each dimensions' length l, binary search is needed to be applied on l^{n-4} arrays. And in worst case it will demand $4l^{n-4}\log_2 l$ comparison. So, we can make the search faster by giving a memory penalty for a *bitmap array* of length $4l^{n-3}$. The *bitmap array* will be a two dimensional array, whose index will represent the history counter value, and one of its entry j (j=1, 2, 3, 4), that will mean dimension of extension and another is a pointer to the EKA(4).

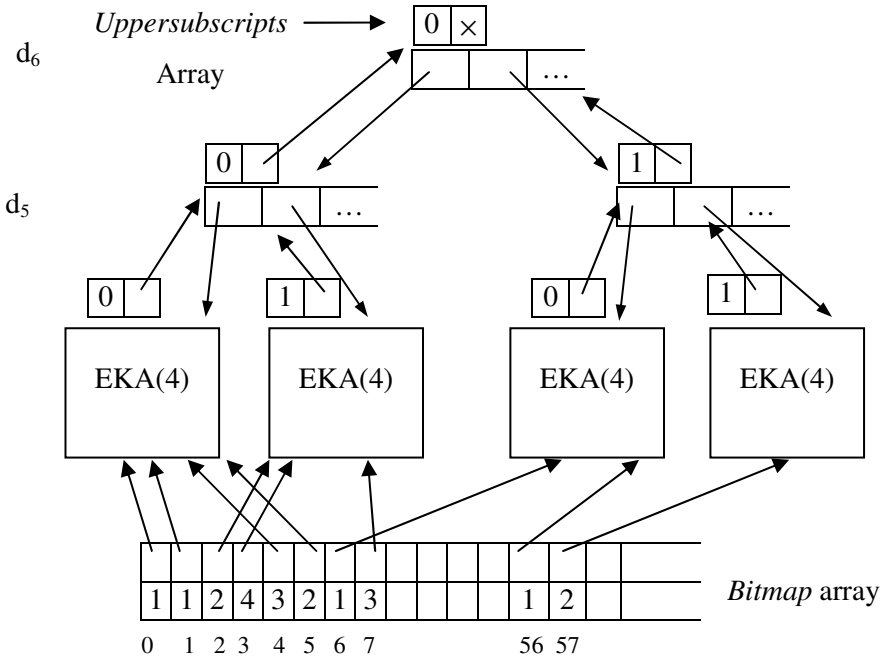


Figure 4.2: Arrangement of HSOC EKA(n) for backward mapping.

Backward Mapping on Compressed EKA(n)

Let, given values are $\langle h,s,o\rangle$. So, we first look at the *bitmap* array at index h and found the entry j and the exact EKA(4) where the h resides. Now apply binary search only over the history table of dimension j H_{dj} to locate the position of h. Now we can determine the lowest 4 dimensions' subscripts by applying the process described in section 4.2.1. Since each EKA(4) maintains a *uppersubscripts* table, the higher dimensional subscripts can be

found from there by going back to root and by collecting *uppersubscripts* array entry. Figure 4.2 shows the logical arrangement of an HSOC EKA(*n*) along with necessary auxiliary tables required for backward mapping.

4.3 Theoretical Analysis

Now we measure the proposed compression scheme on EKA. Before that we present some definitions of basic terms used here as well as some assumptions.

4.3.1 Basic Terms

Density of Array (ρ): it is a parameter to measure the sparsity of an array. It is the ratio of non-empty array cells with total number of cells. Maximum value the density can be one. Formally we can write,

$$\rho = \frac{\text{Total number of cell having non null values}}{\text{Total number of array cells}}$$

Compression Ratio (η): it is defined as the proportionate size of the compressed array with that of uncompressed one, formally

$$\text{Compression ratio, } \eta = \frac{\text{Compressed size of Array}}{\text{Uncompressed size of Array}}$$

Compression ratio value less than one is preferable.

Range of Usability (υ): Range of usability of a compression scheme is defined as the maximum range of data density up to which the compression ratio is less than 1.

4.3.2 Assumptions

All the parameters used in the analysis are given in Table 4.1. in each case, we first present the analysis for EKA(4) and then for EKA(*n*). To make the theoretical analysis simple and tractable, let us consider

- The compressed EKA is extended by a length of one unit in each dimension in round robin manner.
- The length of each dimension is equal i.e. $l_i = l$, for all i.
- All lengths and sizes are in bytes.

Table 4.1: Parameters for compressed EKA.

H_i	History table of dimension i
C_i	Coefficient table of dimension i
E_i	Element table of dimension i
A_i	Address table of dimension i
$N(.)$	Function that returns number elements in an array
$S(.)$	Function that returns size of an array
l_i	Length of each dimension i
α	Size of each offset or auxiliary table cell
β	Size of each cell of the EKA
ρ	Density of the array, $0 \leq \rho \leq 1$
υ	Range of Usability of the array
nEcell	Non empty array cell
Aux	All the auxiliary tables of EKA
P	Higher dimensional pointer arrays

4.3.3 Range of Usability Analysis

For EKA(4)

Number of cell in a *History* table of dimension i (i = 1, 2, 3, 4), $N(H_i) = l$,

Similarly, $N(C_i) = l$, and

$N(E_i) = l$,

Since we consider round robin extension, it can be found that number of cell in a *Address* table,

$$N(A_1 \text{ or } A_2) = 1 + 1 + 2 + 3 + \ldots + l - 1 = \frac{l(l-1)}{2} + 1$$

$$N(A_3 \text{ or } A_4) = 1 + 2 + 3 + \ldots + l = \frac{l(l+1)}{2}$$

Size of History table, $S(H) = \alpha \Sigma_{i=1}^{4} N(H_i) = 4l\alpha$

Similarly, $S(C) = \alpha \ \Sigma_{i=1}^{4} N(C_i) = 4l\alpha$

$S(E) = \alpha \ \Sigma_{i=1}^{4} N(E_i) = 4l\alpha$

$$S(\mathrm{A}) = \alpha \ \Sigma_{i=1}^{4} N(\mathrm{A_i}) = \alpha(l(l-1) + 2 + l\,(l+1)) = \alpha(2l^2 + 2)$$

Therefore, size of auxiliary tables, $S(\mathrm{Aux}) = S(\mathrm{H}) + S(\mathrm{C}) + S(\mathrm{E}) + S(\mathrm{A})$

$$= \alpha(2l^2 + 12l + 2)$$

So, total number of non empty cells in the EKA(4) are, $N(\mathrm{nEcell})$ = density × array size

$$= \rho l^4$$

And total number of offset are, $N(\mathrm{Off}) = \rho\, l^4$.

Then, $S(\mathrm{nEcell}) = \beta N(\mathrm{nEcell}) = \beta\rho\, l^4$

$S(\mathrm{Off}) = \alpha N(\mathrm{Off}) = \alpha\rho\, l^4$

Physical size of the compressed EKA(4),

$$HSOC_{\mathrm{EKA(4)}} = S(\mathrm{Aux}) + S(\mathrm{Val}) + S(\mathrm{Off}) = \alpha(2\, l^2 + 12l + 2) + (\alpha + \beta)\, \rho\, l^4$$

If we would represent the array as traditional representation like TMA, the total number of array cell would be l^4, since array is 4 dimensional and each dimension has a length l. Therefore, physical size of uncompressed Array as TMA, $UC_{\mathrm{TMA(4)}} = l^4\beta$

Compression ratio $\eta = \dfrac{HSOC_{EKA(4)}}{UC_{TMA(4)}} = \dfrac{\alpha(2l^2 + 12l + 2) + (\alpha + \beta)\rho l^4}{l^4\beta}$ ……. (4.1)

To determine Range of Usability, from its definition we can write:

$$\eta = 1$$

$$or, \frac{\alpha(2l^2 + 12l + 2) + (\alpha + \beta)\rho l^4}{l^4\beta} = 1 \qquad [\text{ by eq. (4.1) }]$$

$$or,\ \alpha(2L^2 + 12l + 2) + (\alpha + \beta)\rho l^4 = l^4\beta$$

$$or,\ \rho = \frac{l^4\beta}{(\alpha + \beta)l^4} - \frac{\alpha(2l^2 + 12l + 2)}{(\alpha + \beta)l^4}$$

$$or,\ \rho = \frac{\beta}{\alpha + \beta} - \frac{\beta}{\alpha + \beta}\left(\frac{2}{l^2} + \frac{12}{l^3} + \frac{2}{l^4}\right)$$

$$or,\ \rho \le \frac{\beta}{\alpha + \beta} \qquad \left[\text{Since for large } l,\ \frac{\beta}{\alpha + \beta}\left(\frac{2}{l^2} + \frac{12}{l^3} + \frac{2}{l^4}\right) \cong 0\right]$$

So, we can say, $\upsilon \le \dfrac{\beta}{\alpha + \beta}$

Rather than length of dimension, the range of usability depends on the data type used for the array cell and that of auxiliary tables and offset.

For EKA(*n*)

In EKA(*n*), we have some higher dimensional pointer arrays whose number of cell,

$$N(\mathrm{P}) = \sum_{i=1}^{n-4} l^i = \frac{l^{n-3}-1}{l-1} - 1 \cong O(l^{n-4}), \text{ for large } l, \text{ and } n.$$

Each of these pointer points to an EKA(4).

So, $S(\mathrm{Aux}) = \alpha(2l^2 + 12l + 2)\, l^{n-4} + \alpha l^{n-4} = \alpha(2l^{n-2} + 12l^{n-3} + 3l^{n-4})$

Physical size of the compressed EKA(n),

$$HSOC_{\mathrm{EKA(n)}} = \alpha(2l^{n-2} + 12l^{n-3} + 3l^{n-4}) + (\alpha+\beta)\rho l^n$$

Physical size of uncompressed Array as TMA, $UC_{\mathrm{TMA(n)}} = l^n\beta$

Compression ratio $\eta = \dfrac{HSOC_{EKA(n)}}{UC_{TMA(n)}} = \dfrac{\alpha(2l^{n-2} + 12l^{n-3} + 3l^{n-4}) + (\alpha+\beta)\rho l^n}{l^n\beta}$ …. (4.2)

For EKA(n), we can again find that $\upsilon \le \dfrac{\beta}{\alpha+\beta}$, calculation shown below, which is same as EKA(4). So we can conclude that range of usability is independent of length as well as number of dimension.

$$\eta = 1$$

$$or,\ \rho = \frac{l^n\beta}{(\alpha+\beta)l^n} - \frac{\alpha(2l^{n-2} + 12l^{n-3} + 3l^{n-4})}{(\alpha+\beta)l^n} \quad [\text{ by eq. (4.2) }]$$

$$= \frac{\beta}{(\alpha+\beta)} - \frac{\alpha \times O(l^{n-2})}{(\alpha+\beta)\times O(l^n)}$$

for large l or n, $\dfrac{\alpha \times O(l^{n-2})}{(\alpha+\beta)\times O(l^n)} \cong 0$

So, $\rho \le \dfrac{\beta}{\alpha+\beta}$

It is worth to mention that, above computation excludes the size of auxiliary tables needed for backward mapping. If we consider them then

$$\mathrm{S}(bitmap) = \alpha(2\times 4\times l \times l^{n-4}) = 8\alpha l^{n-3}$$

[Since bitmap array is 2 dimensional, each EKA(4) can have maximum $4l$ history values, and there can be l^{n-4} EKA(4)s.]

$$\text{S}(uppersubscripts) = 2\alpha\sum_{i=1}^{n-4} l^i \cong 2\alpha l^{n-4}, \text{for large } l, n$$

So we can determine that these sizes do not contribute much on auxiliary tables and hence range of usability remains same.

4.3.4 Retrieval Time Analysis

In uncompress EKA the forward mapping time is almost constant, but for a point query in compressed EKA needs $O(\log_2 l^2) = O(2\log_2 l)$ additional time for binary search in a 2-dimensional segment of length l. However, overall range query performance will be better if a sparse array is represented as HSOC EKA rather than straight or uncompressed EKA. This is because in uncompressed form after loading a segment from disk to memory we have to make a linear search to determine the non empty cells. On the other hand in HSOC EKA, we can simply put the segment to display or to any process.

4.4 Conclusion

In this chapter we have presented a compression scheme suitable for EKA. We have also shown that, if we use $\alpha = \beta$, i.e. same data type for the auxiliary tables and the physical array, the scheme can store an array having 50% approximate density. Here it is also presented that usability of the compression scheme doesn't depend on length or number of dimension of the sparse array represented as EKA. In the next chapter we will show the details experimental results that confirm the theoretical analysis presented here as well as in chapter 3.

CHAPTER V

Experimental Analysis

5.1 Experimental Setup

In this chapter, we simulate the retrieval operation for range key query for both TMA and EKA. All lengths or sizes of storage areas are in bytes. Some parameters are provided as input while others are derived from input parameters. We have constructed the TMA and EKA systems having the parameter values shown in Table 5.1 placing the TMA and EKA in secondary storage. The auxiliary tables of EKA are placed in main memory since the sizes of the auxiliary tables are negligible comparing to the main array. The test results for retrieval and extension operations are analyzed in this Section. All the tests are run on a machine (Dell Optiplex 380) of 2.93 GHz processor and 2 GB of main memory having disk page size of 4KByte. We will show that the overall retrieval time has advantages for EKA than TMA. We also show that without any retrieval penalty we can extend the length of dimension of a multidimensional array effectively if implemented using EKA.

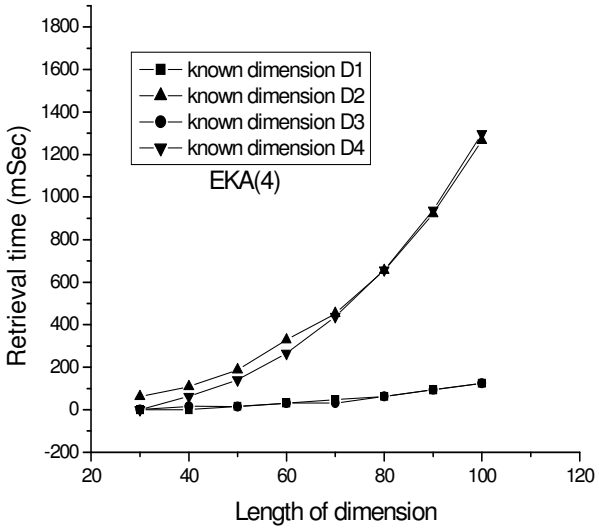
Table 5.1. Assumed parameters for constructed prototypes

n	λ	max(l_i)	Initial V = l^n	NRQ Subscripts
4	10	100	$(30)^4$	(l–λ)/2 to (l+λ)/2
5	5	45	$(20)^5$	
6	2	22	$(10)^6$	

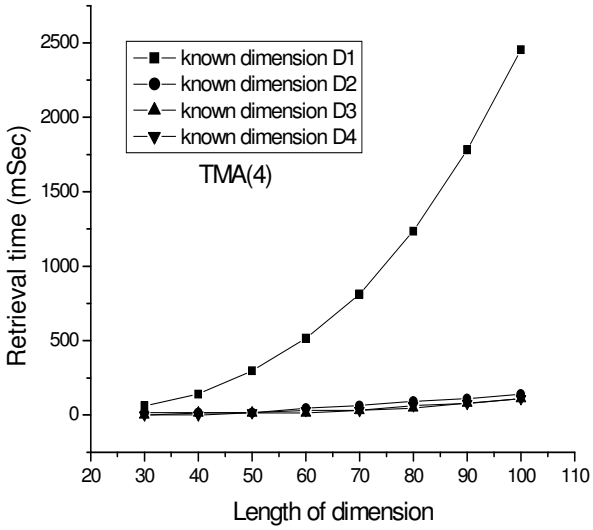
5.2 Experimental Results

5.2.1 Retrieval Cost

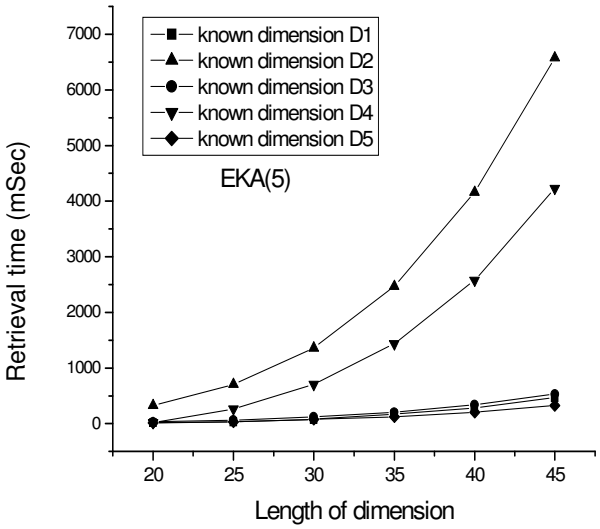
In TMA, the array is linearized in a single data stream using the addressing function; therefore all the offset values of the array elements are consecutive. Hence the range of candidate offset values for a query can be determined uniquely. But for EKA; the same data stream is distributed over different subarrays.



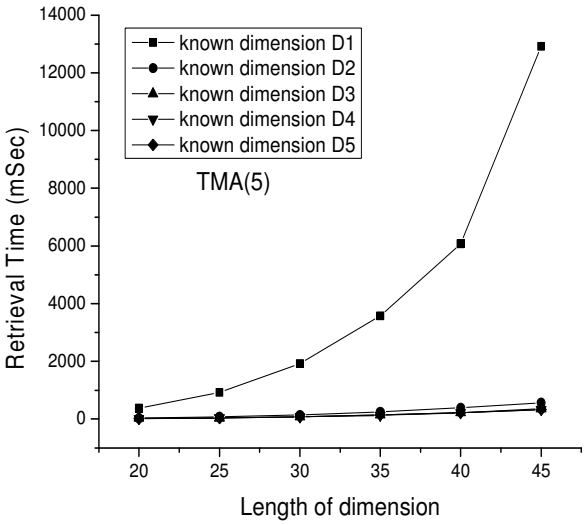
(a) Retrieval Time for EKA(4)



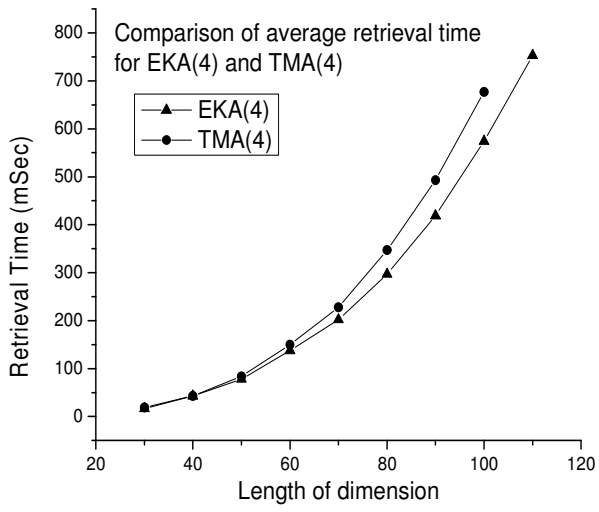
(b) Retrieval Time for TMA(4)



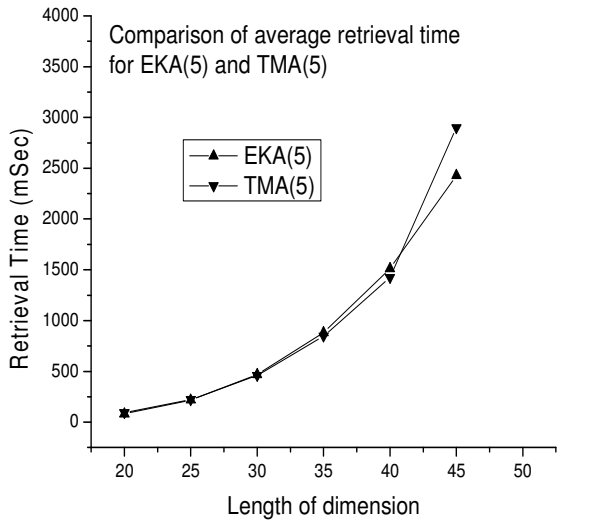
(c) Retrieval Time for EKA(5)



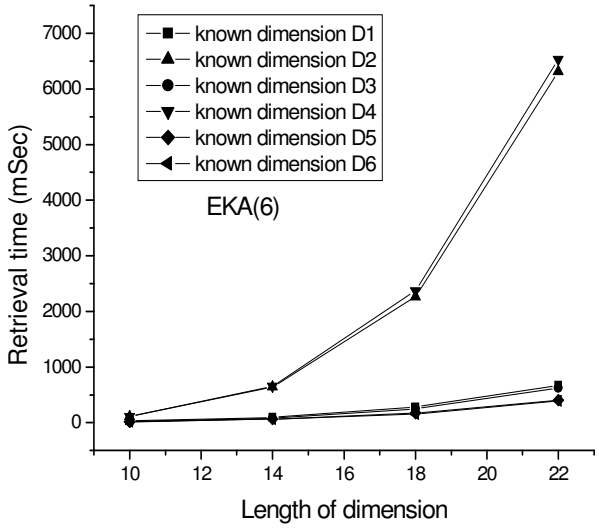
(d) Retrieval Time for TMA(5)



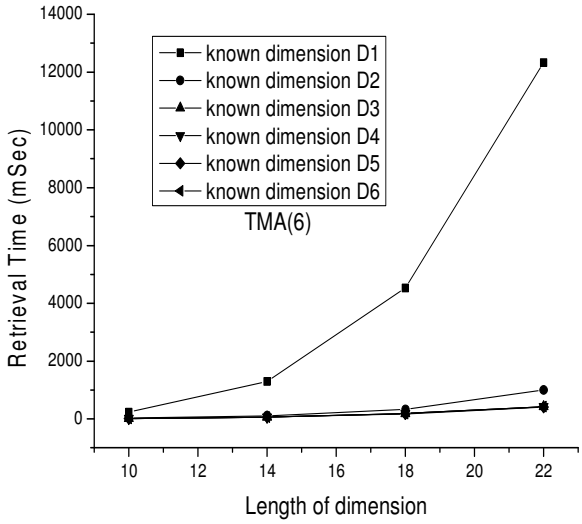
(e) Average retrieval time for EKA(4) and TMA(4)



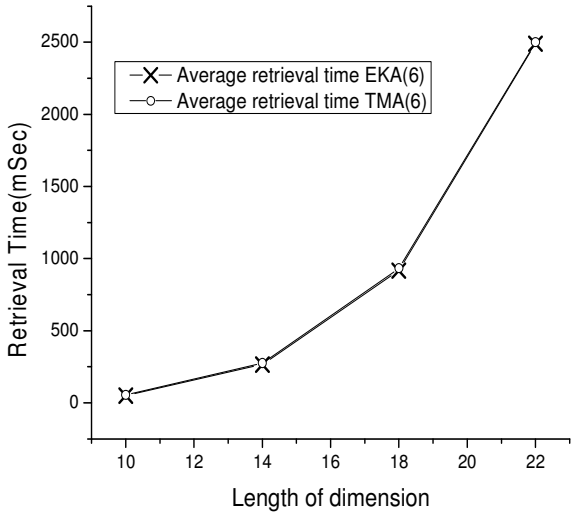
(f) Average retrieval time for EKA(5) and TMA(5)



(g) Retrieval Time for EKA(6)



(h) Retrieval Time for TMA(6)



(i) Average retrieval time for EKA(6) and TMA(6)

Figure 5.1: Retrieval cost analysis for EKA and TMA.

The retrieval performance depends on the known dimension (i.e. the specified dimension) of query dimension. We use the term *known* dimension (or known subscript) to indicate the specified dimension of the query operation. For example if dimension 2 is the known or specified then we write subscript x_2 is *known*.

Figure 5.1 shows the retrieval performance for range key query of TMA and EKA for the parameter values shown in Table 5.1. In Figure 5.1(a) the retrieval performance for EKA(4) for different known dimension is shown. It shows that, the retrieval time is higher when x_2 and x_4 are known. The retrieval time is lower when the x_1 and x_3 is known. This is

because the segments of the subarrays of EKA(4) are two dimensional hence the element inside the subarrays can be organized as row major order or column major order. If the elements are organized in one order (say row major) then it is searched in column order; the target elements for the query are not consecutively organized. Therefore that known subscript takes longer time. Hence two known subscripts will take higher time than other two known subscripts. Please be noted a two dimension array is used as four dimensional (see Figure 3.1) array. When number of dimension n increases for EKA then it (see Figure 5.1(c) and 5.1(g)) shows that retrieval from EKA takes higher time for the known subscripts of only two values. Figure 5.1 (b) shows the retrieval time for TMA for $n = 4$. It shows that for the known dimension of x_1 takes higher time than other known dimensions. This is because when known dimension is x_1 then the entire array needs to be scanned as explained in section 3.6.2.

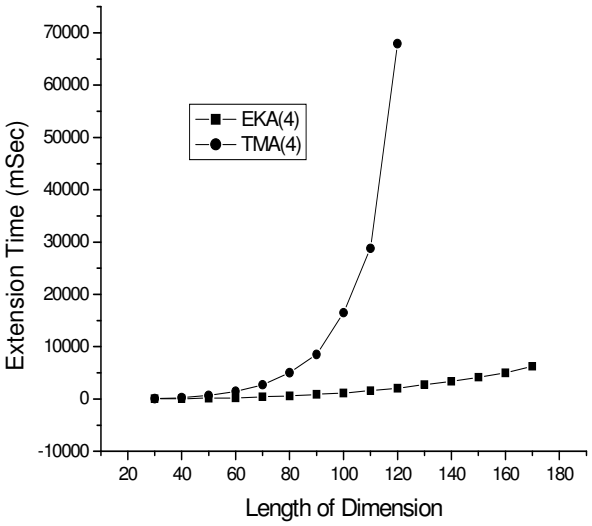
When $n = 5, 6$ for TMA, the same situation i.e. for one known subscript TMA takes higher time than others as shown in Figure 5.1(d) and 5.1(h). Figure 5.1(e) and 5.3(f) shows the average retrieval cost for EKA and TMA for $n = 4$ and 5. It shows that EKA has better performance than TMA and the average retrieval cost is almost same for both EKA and TMA when $n = 6$ (Figure 5.1(i)). It can be concluded that, on average, the retrieval performance for EKA is better and there is no retrieval penalty for EKA over TMA. This conclusion is valid up to $n = 6$, up to which experiment is conducted. For $n > 6$ the performance may or may not deteriorate. We've carried out the experiment up to $n = 6$, since we found it sufficient enough for many practical systems, like MOLAP.

5.2.2 Extension Cost

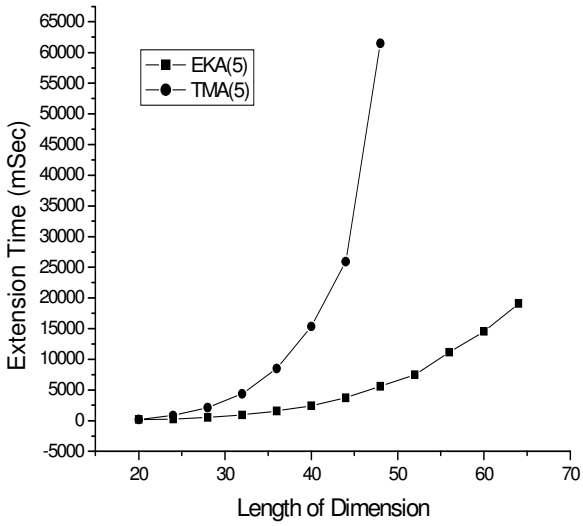
Figure 5.2 shows the extension cost for TMA and EKA .The TMA reorganizes the array whenever there is an extension to it. That is, the whole array will be relinearized on disk to accommodate the new data due to the extension of length of dimension. The TMA scheme needs to face the existing elements then reorganize for the extension. On the other hand, the EKA extends the initial array with segment of subarrays containing the new data as described in Section 3. Hence the EKA strategy can reduce the cost of array extensions significantly.

In Figure 5.2(a), 5.2(b), and 5.2(c) the extension times are shown with $n = 4, 5, 6$, where we find that the extension times for TMA are much higher than EKA. Extension gain is

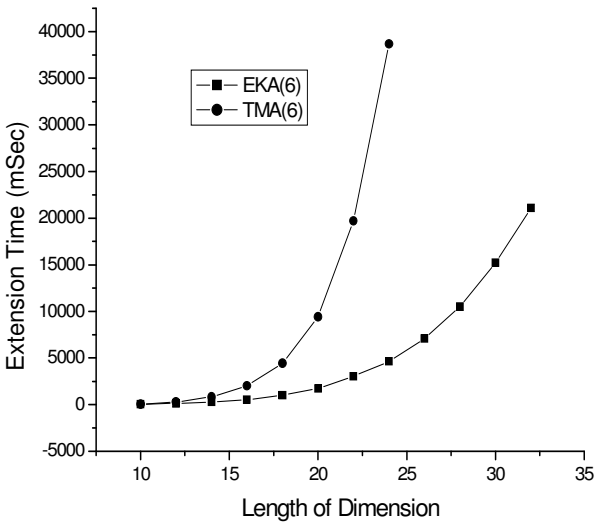
the difference between the extension time of TMA and EKA which is shown in Figure 5.2(d) and 5.2(e). From the theoretical aspects described in section 3.6.3 that the extension is constant with a fixed initial volume for any value of λ. But from Figure 5.2(e) we find that this is almost true for n = 4, but not for others. This is because we made some assumption to make theoretical analysis simple. But in practice we need to populate the auxiliary tables that took some time what we excluded in theory. And populating time increases with large n which affects the extension gain. One other reason is, since λ is variable here, therefore the length of dimension is variable which also affects the lengths of the auxiliary tables as well as the populating time.



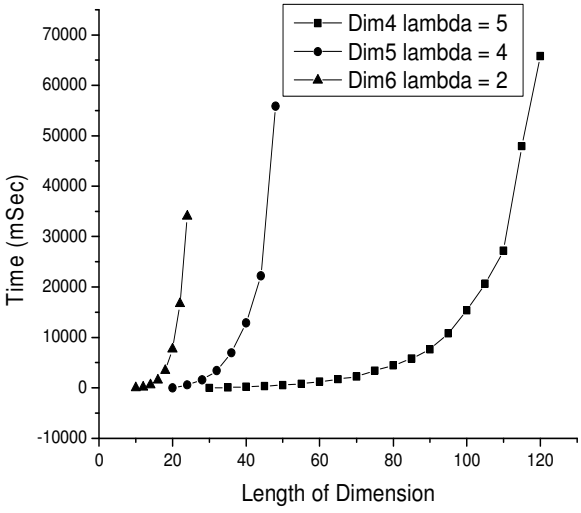
(a) 4-dimensional extension Time



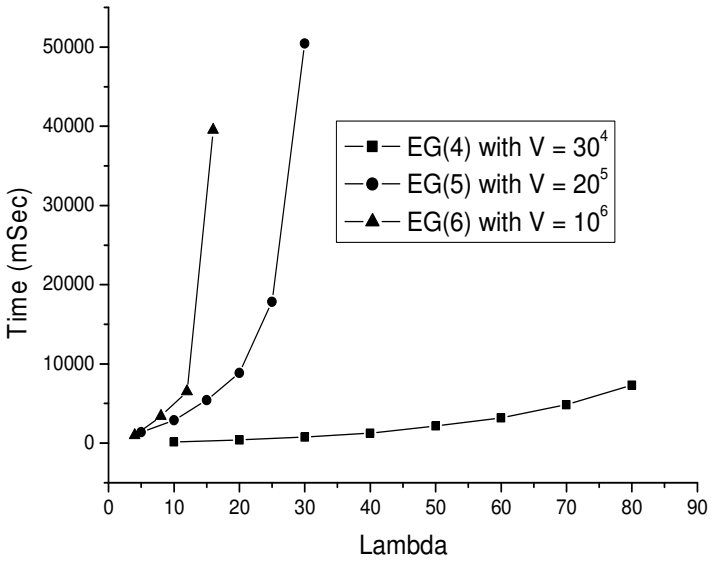
(b) 5-dimensional extension Time



(c) 6-dimensional extension Time



(d) Extension gain with constant λ



(e) Extension gain with constant v

Figure 5.2: Extension cost comparison for EKA and TMA.

The extension cost as well as extension gain depends on the initial volume of the array i.e. the values of n and l before the array is extended. Hence, if n and l increase, then EKA needs less data to store than TMA without any reorganization of data. So TMA needs higher times than EKA and thus gain increases. We can conclude that if the initial volume is large then the extension cost for TMA is higher. It will be expensive to extend a large array even for small values of λ.

5.2.3 Overflow Analysis

Figure 5.3 shows the maximum length of dimension that causes the EKA and TMA to overflow the address space for varying number of dimensions. From Figure 5.3, it is found that, EKA and TMA reaches a length of 180 and 120 respectively in each dimension where for n = 4. Actually EKA doesn't overflow due to memory allocation, it stops allocating secondary storage since the maximum allowable file size is around 4GB for a 32 bit compiler.

Figure 5.4 (a) shows the total storage requirement for EKA and TMA on different number of dimensions varying the length of dimension. From Figure 5.4 (a), it is found that both EKA and TMA need almost same amount of storage up to a particular length of dimension. In practice EKA needs slightly higher amount of storage due to its auxiliary tables, but this is very negligible compared to the total requirement. So we can conclude

that the nature of storage requirement is almost same for EKA and TMA. Figure 5.4 (b) shows the maximum storage allocated for EKA and TMA on different number of dimension before reaching to overflow situation. From Figure 5.4 (b), we find that in all cases EKA allocates storage around 4GB where as TMA allocates around 850 MB. This is because EKA stops on maximum allowable file size, but TMA stops on consecutive memory requirement and/or address space overflow. Though we have 2GB memory TMA can grow only a size of 850MB, this is because - during extension TMA needs almost twice memory space, one space to store the old TMA after reading the data, and another space to allocate for the new TMA after extension.

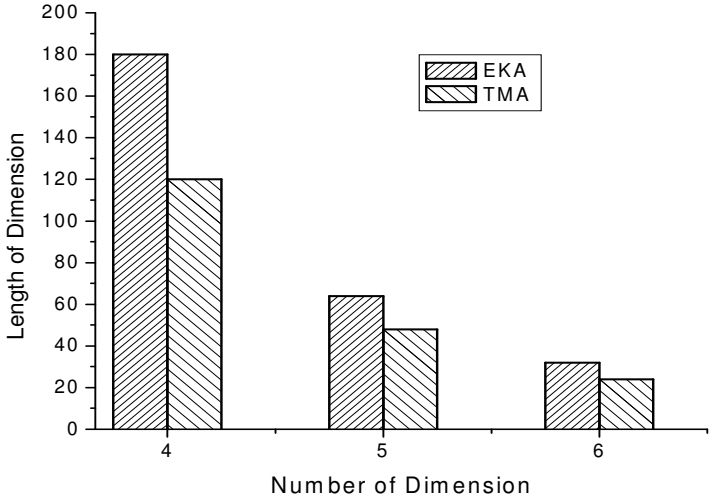
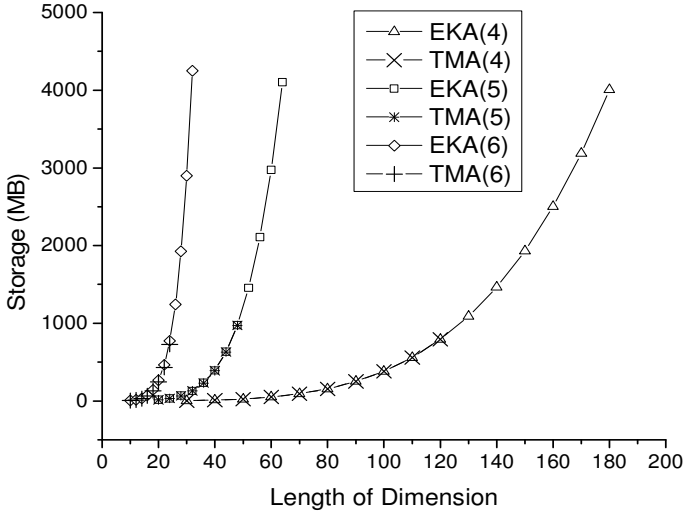
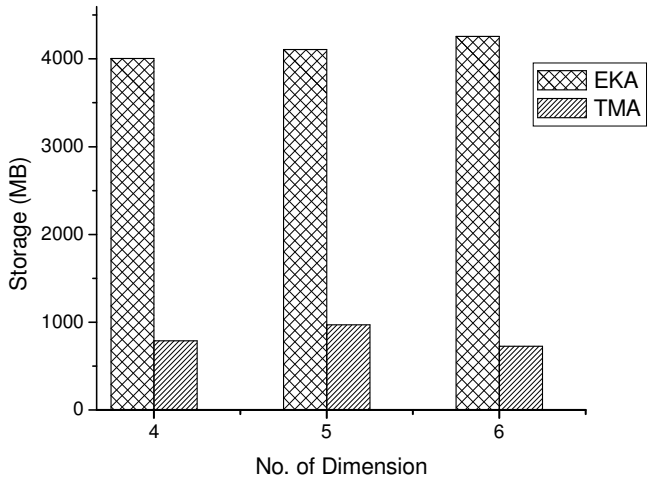


Figure 5.3. Maximum length reached before the occurrence of overflow.



(a) Total storage requirement



(b) Maximum storage allocated before the occurrence of overflow

Figure 5.4: Storage allocation of EKA and TMA.

5.2.4 Compression Results

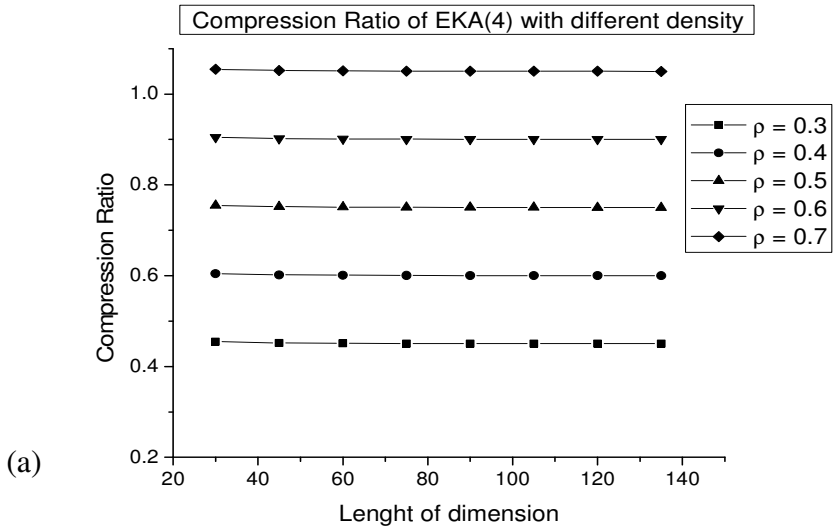
The experimental outcome of compressed EKA is presented in this section. All the simulation is made considering the parameter $\alpha = 4$ bytes, and $\beta = 8$ bytes.

Compression Ratio

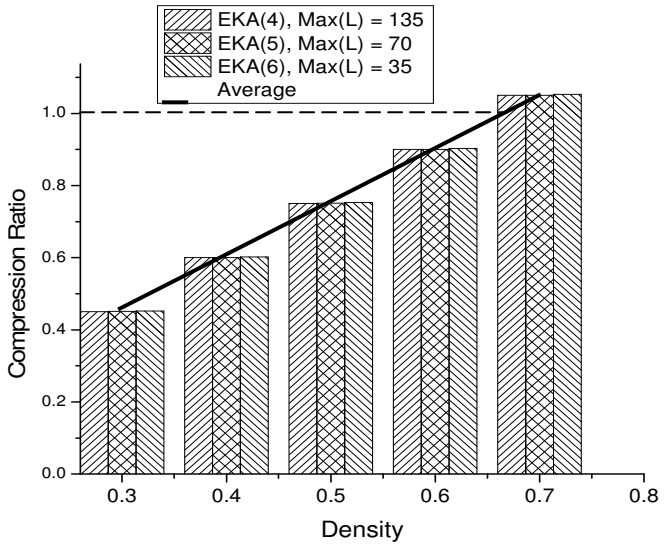
It is an important metric to determine the usability of the compression scheme. Figure 5.5 show the compression ratio found by experimental result of the compression scheme applied on EKA. Figure 5.5(a) shows that compression ratio is almost constant for different length of dimension and density over EKA(4). From Figure 5.5(b) we found the same thing that is compression ratio is independent of different number of dimension, length of dimension and density. In Figure 5.5(b) the line connecting the top of the bars are average compression ratio for different density and it crosses the value one at an approximate density 0.66. Hence range of usability is approximately 0.66 and thus the experimental results proves the theory in section 4.3 (since $\alpha = 4$, and $\beta = 8$).

Extension Time for Compressed EKA

It is another important metric to measure how much time it takes to be extended. Figure 5.6 shows the average extension time measured for $\rho = 0.4$, 0.5, and 0.6 for EKA(4, 5, 6). It shows that extension time exponentially grows with length of dimension, and the growth rate is high for higher number of dimension. This is because for a n-dimensional array, we know the subarray size is l^{n-1}, where l is the length of dimension.



(a)



(b)

Figure 5.5: Compression Ratio for EKA.

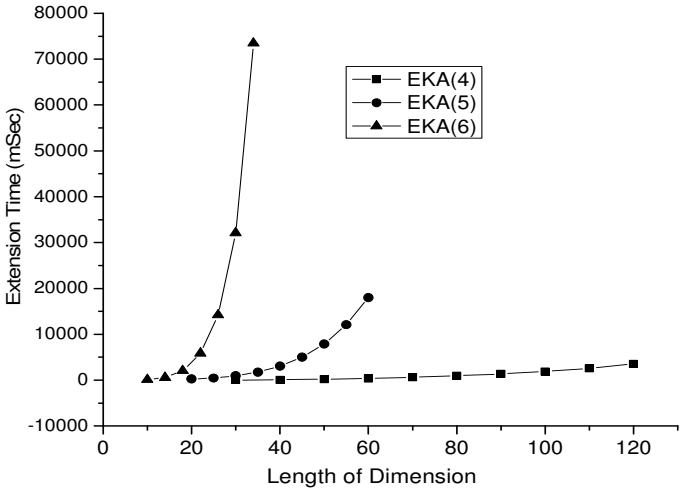
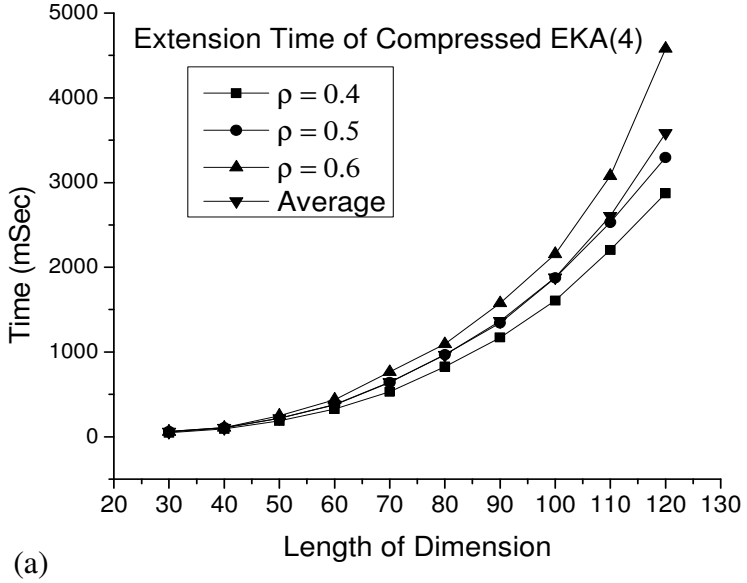
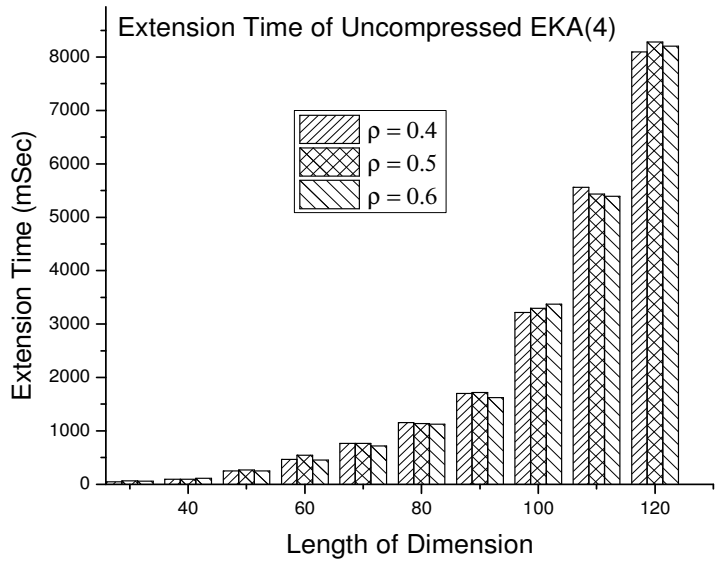


Figure 5.6: Average extension time of compressed EKA.

Figure 5.7 shows the extension time of compressed and uncompressed EKA(4) for different density. For compressed version (Figure 5.7(a)) the extension time varies on density. On the other hand uncompressed EKA(4) always takes almost same amount of time shown in Figure 5.7(b). This is because, in uncompressed version density of real data does not affect the total size of the extension subarray, hence disk I/O is almost constant. In both case the time increases exponentially because the size extension subarray is l^{n-1} which is exponential.



(a)



(b)

Figure 5.7: Extension Time of Compressed and Uncompressed EKA(4).

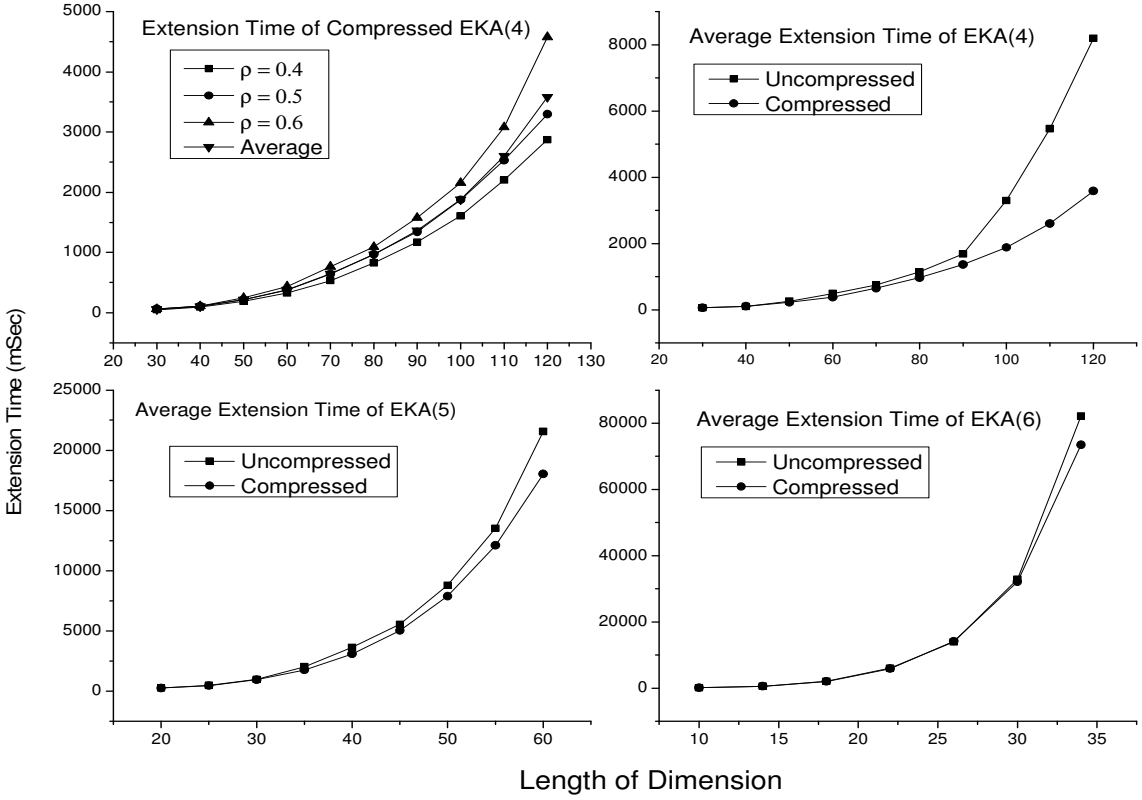
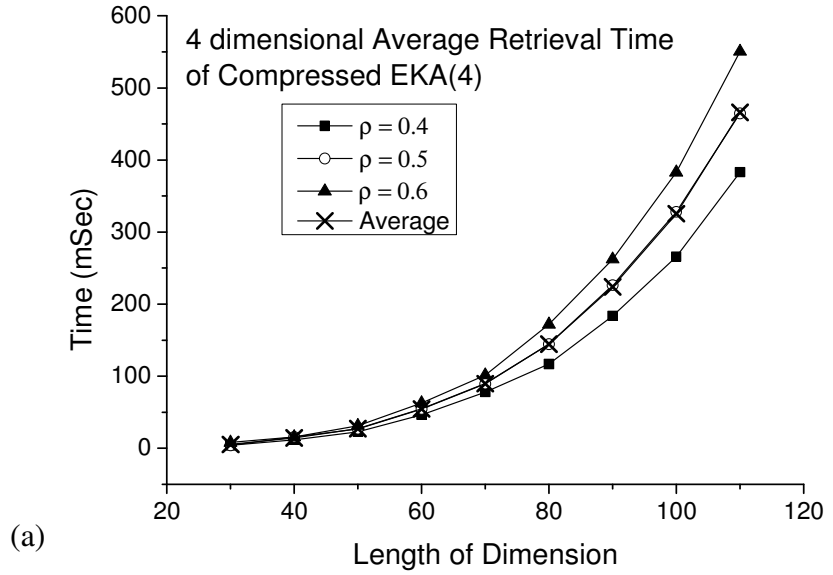


Figure 5.8: Comparison between compressed and uncompressed extension times.

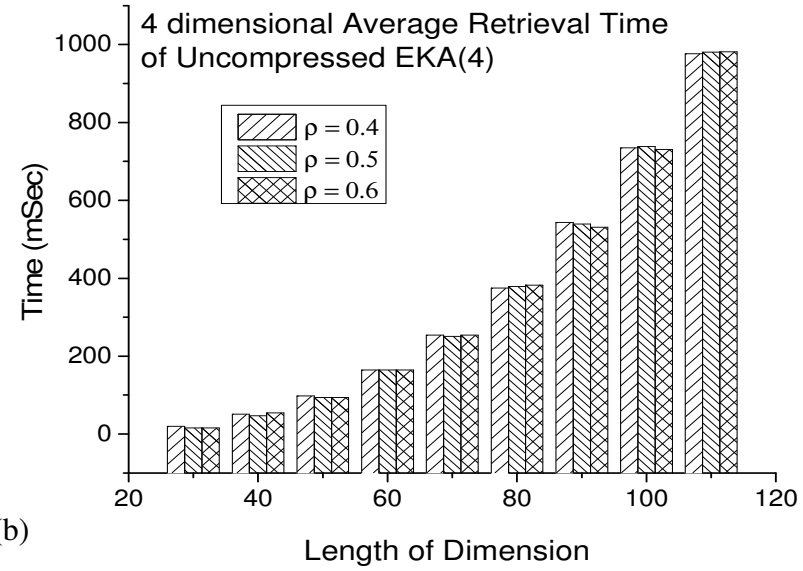
Figure 5.8 shows the average extension time for compressed and uncompressed EKA(4,5,6). From figure we find that in every case compressed version of the array takes less time then uncompressed array. The reason is subtle, compressed array needs less data to write hence fewer disks I/O is required and therefore time is less.

Retrieval Time for Compressed EKA

Figure 5.9 shows the average range key retrieval time of NRQ subscripts on both compressed and uncompressed EKA(4) with different density. Retrieval is made considering each dimension as known dimension and then averaged. From 5.9(a), we find that retrieval time varies with density in compressed EKA(4). However there is no effect of density in uncompressed one, the retrieval time is almost constant for a particular length of dimension (see Figure 5.9(b)). This is because in uncompressed EKA(4) whatever the density, the segment or subarray sizes remain same, hence retrieval time is constant. Though we have presented only EKA(4), we found same phenomena for EKA(5) and EKA(6) also.



(a)



(b)

Figure 5.9: Average range key retrieval on compressed and uncompressed EKA(4).

The retrieval time linearly increases with the change of density considering a constant length of EKA. Figure 5.10 exhibits this feature on EKA(5) with different length. The reason is, for an n-dimensional array with a particular length l and density ρ the number of non empty cell is ρl^n. So if ρ changes the total number changes linearly and hence the retrieval time.

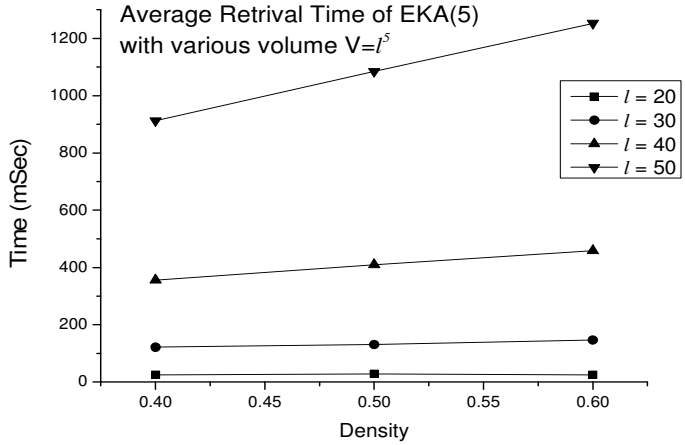


Figure 5.10: Change of retrieval time with density in compressed EKA(5).

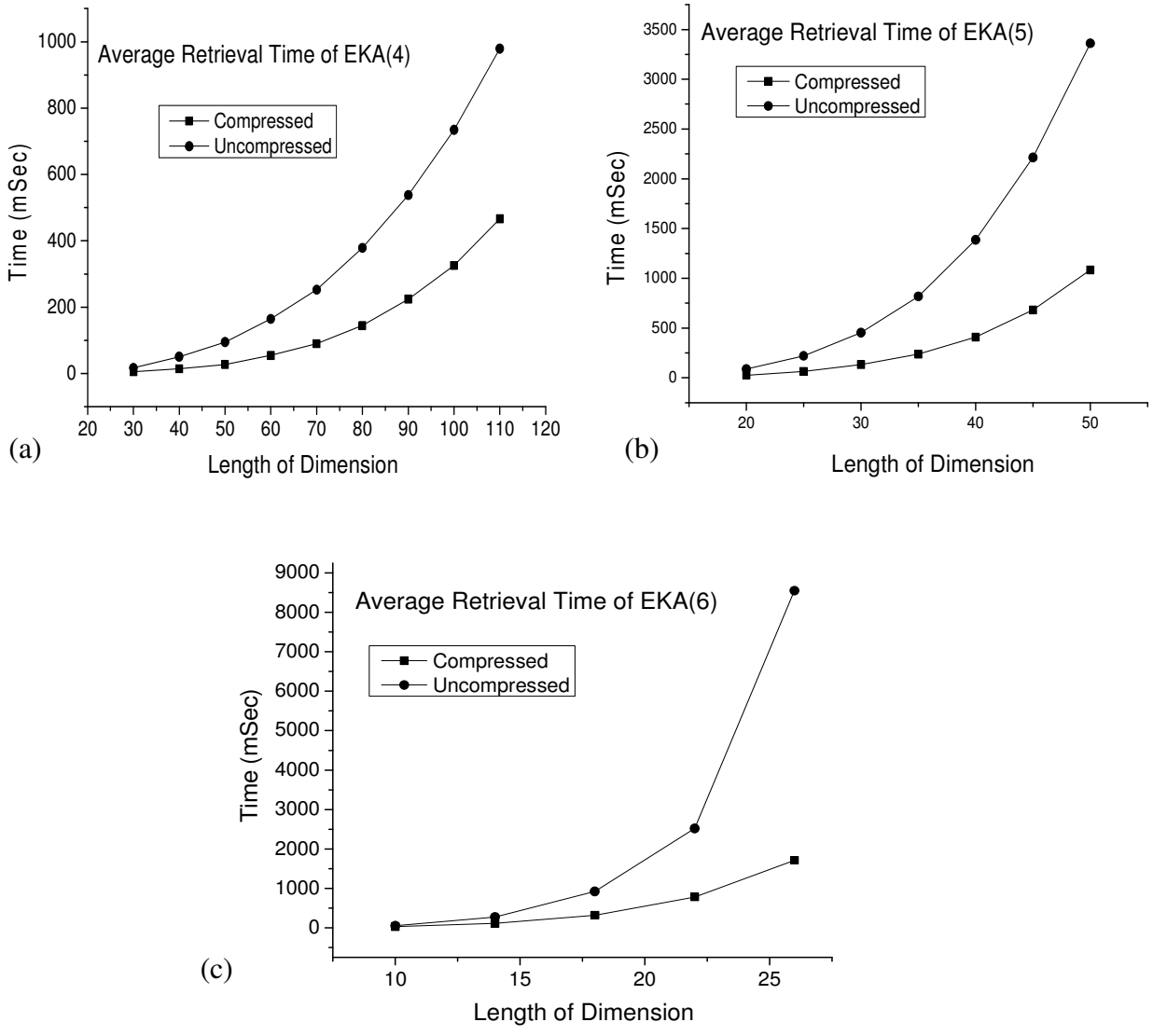


Figure 5.11: Average retrieval time comparison between compressed and uncompressed EKA.

Figure 5.11(a), 5.11(b), and 5.11(c) show the comparison of range key retrieval time of NRQ subscripts in compressed and uncompressed EKA(4), EKA(5), and EKA(6) respectively. Here given retrieval time is the average of retrieval time with density $\rho = 0.4$, 0.5, and 0.6. However the retrieval time with a density is, in fact, an average retrieval time considering each dimension as known dimension. In every case the compressed EKA needs much less time than uncompressed one, which is depicted in Figure 5.11. The reason is, for a range key query we have to determine major and minor subarray and then load the subarray or segment from disk to memory. In uncompressed EKA whatever the density segment size is always same and maximum. Furthermore if density is less than 1, we need a linear search to be made for determining the non empty cells. But in compressed EKA the segments are compact and their size varies with density. Since the segments contain only the non empty cells of the logical array there is no need of any search. Simply read the segment from disk and present them, which require much less time. The same thing is true for segments other than major or minor subarray. Therefore overall retrieval time in compressed EKA is better than uncompressed EKA.

5.3 Discussion

In this chapter we present the experimental outcomes of the proposed scheme. We compare the various operations on proposed scheme with that of TMA. We also make comparison between compressed and uncompressed version of the proposed model. In each case we found relevancy with the theoretical analysis what we made in Chapter III and IV. We find that EKA outperform TMA for retrieval and extension operation, and furthermore compressed EKA is better than that of uncompressed EKA.

CHAPTER VI

Conclusion

6.1 Summary

Many scientific applications extensively use multidimensional array to represent their data for efficient processing purpose. However in many cases the total number of data or dimension cannot be predicted beforehand. Besides this, representing the real world data in multidimensional array creates a very sparse array. In this research work, we managed three practical problem of multidimensional data representation namely (i) extending the length or size of the array dynamically, (ii) address space overflow, and (iii) sparsity of array.

We proposed a new scheme namely Extendible Karnaugh Array (EKA) for multidimensional array representation. The main idea of the proposed model is to represent multidimensional array by a set of two dimensional extendible arrays. To extend the TMA re-linearization is necessary but this is very costly when the array is large. Therefore we need an array system to extend in all dimensions without costly shuffling of the existing data. Our proposed EKA model serves this purpose efficiently. Most of array systems do not consider the address space overflow problem, but proposed scheme manages the problem effectively. Sparsity creates a new direction of data representation, so to handle it we also presented a suitable compression scheme for the proposed model.

We evaluated the proposed EKA and its variant i.e. the compressed EKA by theory and experiment. The experimental results confirm the theory for various array operations. Again we compared the EKA with traditional array representation and found better results for the proposed model.

6.2 Future Scope of Work

Though the overall effect will very little, but there is still a scope to minimize the auxiliary tables maintained in proposed EKA. Since the proposed model is a multidimensional array

representation scheme, any application or system that uses multidimensional array to represent data can use the scheme. More specifically –

- This scheme can be successfully applied to database applications especially for multidimensional database or multidimensional data warehousing system.
- One important future direction of the work is that, the scheme can be easily implemented in parallel platform.
- Because most of the operations described here is independent to each other. Hence it will be very efficient to apply this scheme in parallel and multiprocessor environment.

REFERENCES

1. Seamons, K.E. and Winslett, M., 1994, "Physical Schemas for Large Multidimensional Arrays in Scientific Computing Applications", Proc. of 7th International Conference on Scientific and Statistical Database Management (SSDBM), pp. 218–227, IEEE CS, Washington, DC, USA.
2. Sarawagi, S. and Stonebraker, M.,1994, "Efficient Organization of Large multidimensional Arrays", Proc. of 10th International Conference on Data Engineering, pp. 328–336, Houston, TX , USA.
3. Li, J. and Srivastava, J., 2002, "Efficient Aggregation Algorithms for Compressed Data Warehouses", IEEE Transaction on Knowledge and Data Engineering, Vol. 14, No. 3, pp. 515–529.
4. Zhao, Y., Deshpande, P.M. and Naughton, J. F., 1997, "An Array Based Algorithm for Simultaneous Multidimensional Aggregates", ACM SIGMOD, pp. 159–170.
5. Acker, R., Pieringer, R. and Bayer, R., 2005, "Towards Truly Extensible Database Systems", Proc. of DEXA conference, LNCS, Vol. 3588, pp. 596–605.
6. Hasan, K.M.A., Azuma, M.N., Tsuji, T., and Higuchi, K., 2005, "An Extendible Array Based Implementation of Relational Tables for Multidimensional Databases", Proc. of DaWak, LNCS, Vol. 3580, pp. 233–242.
7. Otoo, E. J. and Merrett, T.H., 1983, "A Storage Scheme for Extendible Arrays", Computing, Vol. 31, pp. 1–9.
8. Sidiroglou, S., Giovanidis, G. and Keromytis, A. D., 2005, "A Dynamic Mechanism for Recovering from Buffer Overflow Attacks", Information Security Conference/Information Security Workshop - ISC(ISW), pp. 1–15.
9. Chiueh, T. and Hsu, F., 2001, "RAD: A Compile-Time Solution to Buffer Overflow Attacks," Proc. of ICDCS, pp. 409–417.
10. Hasan, K.M.A., Tsuji, T. and Higuchi, K., 2006, "A Parallel Implementation Scheme of Relational Tables Based on Multidimensional Extendible Array", Journal of Data Warehousing and Mining, Vol. 2, No. 4, pp. 66–85.
11. Chun, Y. L., Jen, S.L. and Yeh, C.C., 2002, "Efficient Representation Scheme for Multidimensional Array Operations," IEEE Transactions on Computers, Vol. 51, No. 3, pp. 327–354.

12. Kumakiri, M., Bei, L., Tsuji, T. and Higuchi, K., 2006, “Flexibly Resizable Multidimensional Arrays”, Proc. of 22nd International Conference on Data Engineering Workshops, pp. 83−88, Atlanta, GA, USA.

13. Chun, Y. L., Yeh, C.C. and Jen, S.L., 2003, “Efficient Data Compression Methods for Multidimensional Sparse Array Operations Based on the EKMR Scheme,” IEEE Transactions on Computers, Vol. 52, No. 12, pp. 1640−1646.

14. Rosenberg, A.L., 1974, “Allocating Storage for Extendible Arrays”. Journal of the ACM (JACM), Vol. 21, pp. 652−670.

15. Rosenberg, L. and Stockmeyer, L. J., 1977, “Hashing Schemes for Extendible Arrays”, JACM, Vol. 24, pp.199−221.

16. Otoo, E. J. and Rotem, D., 2006, “A Storage Scheme for Multi-dimensional Databases Using Extendible Array Files”, Proc. of 3rd Workshop on STDBM, pp. 67−76, Seoul, Korea.

17. Otoo, E. J. and Rotem, D., 2006, “Efficient Storage Allocation of Large-Scale Extendible Multi-dimensional Scientific Datasets”, Proc. of 18th International Conference on SSDBM, pp. 179−183, Vienna, Austria.

18. Kumakiri, M., Li, B., Tsuji, T. and Higuchi, K., 2006, “Flexibly Resizable Multidimensional Arrays”, Proc. of 22nd International Conference on Data Engineering Workshops, pp. 83−88, Atlanta, GA, USA.

19. Li, B., Tsuji, T. and Higuchi, K., 2007, “Sharing Flexibly Resizable Multidimensional Arrays in Client/Server Environment” Proc. of the International Workshop on Databases for Next Generation Researchers, pp. 19−24, Istanbul.

20. Rotem, D. and Zhao, J.L., 1996, “Extendible Arrays for Statistical Databases and OLAP Applications”, Proc. of 8th International Conference on SSDBM, pp. 108−117, Stockholm, Sweden.

21. Barret R., Berry M., Chan T.F., Dongara J., Eljkhhout V., Pozo R., Romine C. and Van H., 1994, “Templates for the Solution of Linear Systems: Building Blocks for the Iterative Methods”, SIAM, 2nd. ed.

22. White J. B. and Sadayappan P., 1997, “On Improving the Performance of Sparse Matrixvector Multiplication”, Proc. of International Conference on High Performance Computing, pp. 711−725.

23. Shao, Y., Deshpande, P.M. and Naughton, J.f., 1997, “An Array Based Algorithm for Simultaneous Multidimensional Aggregate”, Proc. of SIGMOD’97, pp. 159−170.

24. Tsuji, T., Hara, A. and Higuchi, K., 2006, “An Extendible Multidimensional Array System for MOLAP”, SAC’06 April pp. 23−27, Dijon, France.

25. Shimada, T., Fang, T., Tsuji, T. and Higuchi, K., 2006, “Containerization Algorithms for Multidimensional Arrays”, Asia Simulation Conference, pp. 228−232, Heidelberg: Springer-Verlag.

26. Tsuji, T., Jin, D. and Higuchi, K., 2008, “Data Compression for Incremental Data Cube Maintenance”, DASFAA, LNCS, Vol. 4947, pp. 682−685.

27. Mano, M.M., 2005, “Digital Logic and Computer Design”, Prentice Hall.

28. Pedersen, T. B. and Jensen, C. S., 2001, “Multidimensional Database Technology”, IEEE Computer, Vol. 34, No.12, pp. 40−46.

29. Rotem, D., Otoo, E. J. and Seshadri, S., 2007, “Optimal Chunking of Large Multidimensional Arrays for Data Warehousing”, Lawrence Berkeley National Laboratory, University of California, LBNL-63230.

30. Stabno, M. and Wrembel, R., 2007, “RLH: bitmap compression technique based on run-length and huffman encoding”, Proc. of the 10th International Workshop on Data Warehousing and OLAP (DOLAP ’07), pp. 41−48.

31. Stockinger, K. and Wu, K., 2007, “Bitmap indices for data warehouses”, In Wrembel, R. and Koncilia, C. (eds), DataWarehouses and OLAP: Concepts, Architectures and Solutions, Idea Group Inc. ’07, ISBN 1-59904-364

32. Eggers, S. and Shohani, A., 1980, “Efficient Access of Compressed Data”, Proc. of 6th International Conference on Very large Databases, pp. 205−211.

33. Kornacker, M., 1999, “High-Performance Extensible Indexing”, Proc. of VLDB, pp. 499−508.

34. D. Knuth, 1973, “The Art of Computer Programming”, Vol. 3: Sorting and Searching, Addison-Wesley Publ. Co, Reading, Mass.

35. Bertino, E. and Kim, W., 1989, “Indexing Techniques for Queries on Nested Objects”, IEEE Transactions on Knowledge and Data Engineering, Vol. 1, No. 2, pp. 196−214.

36. Jin, D., Tsuji, T., Tsuchida, T. and Higuchi, K., 2008, “An Incremental Maintenance Scheme of Data Cubes”, Proc. of DASFAA, pp. 172−187.

37. Apaydin, T., Canahuate, G., Ferhatosmanoglu, H. and Tosun A. S., 2006, “Approximate Encoding for Direct Access and Query Processing over Compressed Bitmaps”, Proc. of Conference on Very Large DataBases (VLDB), pp. 846–857.

38. Bassiouni, M. A., 1985, “Data Compression in Scientific and Statistical Databases”, IEEE Transaction on Software Engineering, Vol. 11, No.10, pp. 1047−1057.

39. Datta, A. and Thomas, H., 2002, “Querying Compressed Data in Data Warehouses”, Journal of Information Technology and Management, Vol. 3, No. 4, pp. 353−386.

40. Li, J., Rotem, D. and Wong, H. K., 1987, “A New Compression Method with Fast Searching on Large Databases”, Proc. of 13th International Conference on Very Large Databases, pp. 311−318, Morgan Kaufman.

41. Kim, M. H. and Lee, K. Y., 2006, “Efficient Incremental Maintenance of Data Cubes”, Proc. of 32nd International Conference on Very Large Databases, pp. 823–833, Morgan Kaufman.

42. Ng, W. and Chinya, V. R., 1997, “Block-Oriented Compression Techniques for Large Statistical Databases”, IEEE Transaction on Knowledge and Data Engineering, Vol. 9, No. 2, pp. 314–328.

43. Owen, K., 2002, “Compressing MOLAP Arrays by Attribute-Value Reordering: An Experimental Analysis”, UNBSJ ASCS Technical Report TR-02-001.

44. Hasan, K.M. A., Tsuji, T., and Higuchi, K., 2007, “An Efficient Implementation for MOLAP Basic Data Structure and Its Evaluation”, Proc. of DASFAA, LNCS, Vol. 4443, pp. 288–299.

45. Tsuji, T., Kuroda, M. and Higuchi, K., 2008, “History Offset Implementation Scheme for Large Scale Multidimensional Data Sets,” Proc. of ACM Symposium on Applied computing, pp. 1021−1028.