An Efficient Encoding Scheme to Handle the Address Space Overflow for Large Multidimensional Arrays

Sk. Md. Masudul Ahsan, K. M. Azharul Hasan
Department of Computer Science and Engineering
Khulna University of Engineering & Technology (KUET)
Khulna 9203, Bangladesh
Email: smmasudulahsan@yahoo.com, azhasan@gmail.com

***Abstract* - We present a new implementation scheme of multidimensional array for handling large scale high dimensional datasets that grows incrementally. The scheme implements a dynamic multidimensional extendible array employing a set of two dimensional extendible arrays. The multidimensional arrays provide many advantages but it has some problems as well. The Traditional Multidimensional array is not dynamic extendible. Again, if the length of dimension and number of dimension of a multidimensional array is large then the address space for the array overflows quickly. In this paper, we propose a solution against the essential problem of address space overflow for handling large scale multidimensional datasets using our implementation model. We also propose a record encoding scheme based on our model for representing relational tables using multidimensional array. We evaluate our proposed scheme by comparing with Traditional Multidimensional Array (TMA) for different operations and find a reasonable delay of address space overflow with no retrieval penalty. We also compare the encoded scheme with traditional scheme and find that proposed encoded scheme performs better on range retrieval for sparse array.**

***Index Terms* - Multidimensional Array, Extendible Array, Address space overflow, Karnaugh Map, MOLAP, Dynamic Extension and Sparse Array.**

I. INTRODUCTION

Large scale scientific and engineering data are often modeled in multidimensional arrays [1-3]. The Traditional Multidimensional Array (TMA) based array files are used for storing such large datasets [4]. TMA is an efficient organization in terms of accessing the element of the array by straight computation of the addressing function, but it is not dynamically extendible [5,6]. One more problem with the TMA is that the address space overflows quickly when the length of dimension and number of dimension is large [7]. There are some extendible data structures [4,5,8] that can be extended dynamically but they also overflow for large number of dimensions or length of dimension. In this paper we propose a new scheme namely Extendible Karnaugh Array (EKA), to represent the multidimensional data. EKA has the property of dynamic extension during run time and significantly delays the occurrence of address space overflow. The main idea of EKA lies in representation of an n dimensional array by a set of two dimensional extendible arrays[9].

An n dimensional 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 $0 \le j_1 < l_1$, $0 \le j_2 < l_2$, …, $0 \le j_n < l_n$, there corresponds an element of E. The domain from which the elements are chosen is immaterial and we make the assumption that k bytes of storage area are needed to each n-tuples. The set of continuous memory locations into which the array maps is denoted by $A[0{:}D]$, where $D = \prod_{1 \le i \le n} l_i - 1$ and the size of the array is denoted by $S = D \times k$. If number of dimension n and the length of each dimension l_i $(1 \le i \le n)$ increase the total address space or array size S becomes very large that causes to overflow the existing data types even for 64 bit machines. Hence it is impossible to allocate such a large size multidimensional array.

We overcome this problem by employing a new implementation structure called Extendible Karnaugh multidimensional Array (EKA). EKA is based on multidimensional extendible array [4,5,7]. An extendible array can be extended dynamically without reallocating the data that already exists. It stores an array and all its potential extensions. We show that without any retrieval penalty the array can be extended dynamically in any direction. The scheme is fully general to be applied in different applications such as implementation of buffer overflow attacks [10,11] and multidimensional database systems [6,12].

The paper is organized as follows: in Section II, we discuss about the existing related works in this domain, the basic EKA structure and its operations are presented in section III. Section IV describes the proposed History Segment-Offset Encoding (HSOE) scheme; Experimental results are explained in Section V. And finally Section VI concludes the paper.

doi:10.4304/jcp.8.5.1136-1144

II. Related Works

Some existing multidimensional data representation systems are Traditional Multidimensional Array (TMA) [1,2,3], Extended Karnaugh Map Representation (EKMR) [13,14], Extendible multidimensional Array [4,5,7]. TMA is a fine storage of multidimensional data that represents *n* dimensional data by an array cell in an *n* dimensional array. To insert a new column value in the TMA entire reorganization of the array is necessary which causes massive cost [8]. The EKMR represent *n* dimensional data by a set of two dimensional arrays. But EKMR is not dynamically extendible. The Extendible Array [4,5,7] has the property of extendibility without reorganizing previously allocated elements. Although it can be extended dynamically, it will overflow quickly for address space because its subarrays are *n*-1 dimensional. An extendible array model [15] where there is a record for each dimension called axis vector. In this approach for two consecutive extensions along the same dimension, there is 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. However, for round robin expansion it is similar to extendible array [4], and have same address space overflow problem. [3,16] presents the chunking of multidimensional arrays. In this scheme the large multidimensional arrays are broken into chunks for storage and processing. All the chunks are *n* dimensional with smaller length than the original array. But the dynamic extension is not possible. One more problem is that binary search is necessary to locate a chunk [17]. All the array models mentioned in this section do not handle the address space overflow which is handled in our EKA scheme.

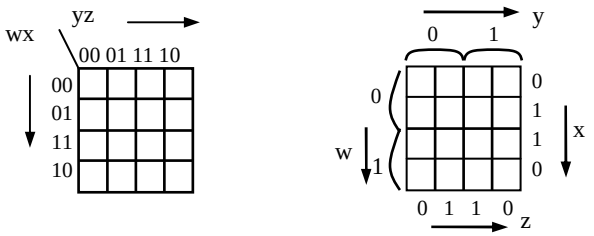
A cell in a array can be encoded using its subscripts [1,2]. For example the subscript $\langle x_1, x_2, \ldots, x_n \rangle$ can be used to refer a cell of the array but if the number of dimension *n* increase then storage requirement will be higher. Also dynamic handling will be necessary for variable values of *n*. An offset value can be used for TMA to refer a cell in an array[1,2,8]. The offset can be calculated using the addressing function. This offset will overflow for large values of *n* and length of dimension. The history-offset encoding [6,7,18] can handle the dynamic situation based on extendible array but it also overflows very quickly as the subscripts are *n*-1 dimension. The chunk-offset encoding [19,20] is based on chunk number and offset inside the chunk. Each of the chunks is an *n* dimensional array but length of dimension is smaller than the original array. To handle the overflow the length of dimension becomes very small for large value of *n*. one more problem of the scheme is that the dynamic extension of the array is not handled in chunk offset encoding. Although ⟨offset⟩, ⟨history, offset⟩, and ⟨chunk, offset⟩ scheme requires fixed amount of memory for different values of *n* but it is difficult to use them in actual implementation when overflow situation is concerned.

An array encoding scheme based on EKMR is proposed in [13,14]. The scheme uses $I = i \times j$, $J = k \times l$. But it also uses a TMA as the basic data structure and overflow situation is same as TMA. In our ⟨history, segment, offset⟩ encoding the memory requirement is fixed for $n \leq 4$, and the dynamic situation is handled efficiently. But when $n > 4$, then extra subscripts will be needed and memory requirement will not be fixed.

Many of them [4,5,17,18] have a concept of subarray which is *n*-1 dimensional if the array having *n* dimensions. Maximum value of coefficient vector is fairly large even for *n*-1 dimensional subarray, and quickly overflows. On the other hand, the proposed EKA [21] is a set of two dimensional arrays; therefore maximum value of coefficient vector is relatively small even for large number of dimensions. And hence, delays the occurrence of overflow. The retrieval performance of EKA out performs the traditional multidimensional array (e.g. TMA), and EKA has dynamic behavior. Some other existing schemes like CRS/CCS or ECRS/ECCS [22-24] works very well for a static array, but they cannot grow or extend in runtime. The proposed HSOE encoding method suitably manages the problems.

III. The Extendible Karnaugh Array

The idea of EKA is based on Karnaugh Map (K-map) [25]. The detail of the EKA can be found in [9,21]. The K-map technique is used for minimizing Boolean expressions usually aided by mapping values for all possible combinations. Fig. 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 of K-map

Figure 1. Realization of Boolean function using K-map

The array representation of a K-map for 4 variable Boolean function is shown in Fig. 1(b). The length of each of the dimensions is 2 for both Fig. 1(a) and (b). This is because the Boolean variables are binary that causes the length to be 2.

Definition 1 (Adjacent Dimension): 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(a) and (b) i.e. *adj*(w) = x or *adj*(x) = w.

EKA is the combination of subarrays. It has three types of auxiliary tables namely *history table, coefficient table*, and *address table*. For each dimension these tables

exist. These tables help the elements of the EKA to be accessed very fast.

Any element in the *n* dimensional array is determined by an addressing function as follows,

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

The coefficients of the addressing function namely $\langle l_1 l_2 ... l_{n-1}, l_1 l_2 ... l_{n-2}, ..., l_1 \rangle$ are referred to as coefficient vector which are stored in coefficient table.

The extension subarray is divided into equal size parts known as segment that can be stored contiguously on disk. The number of segments determines the number of entries in the address table and is calculated from the length of adjacent dimension. The first address of a segment can be used to compute the correct position of an element. The segments are always 2 dimensional for an *n* dimensional EKA, EKA(*n*).

A. 4 Dimensional EKA

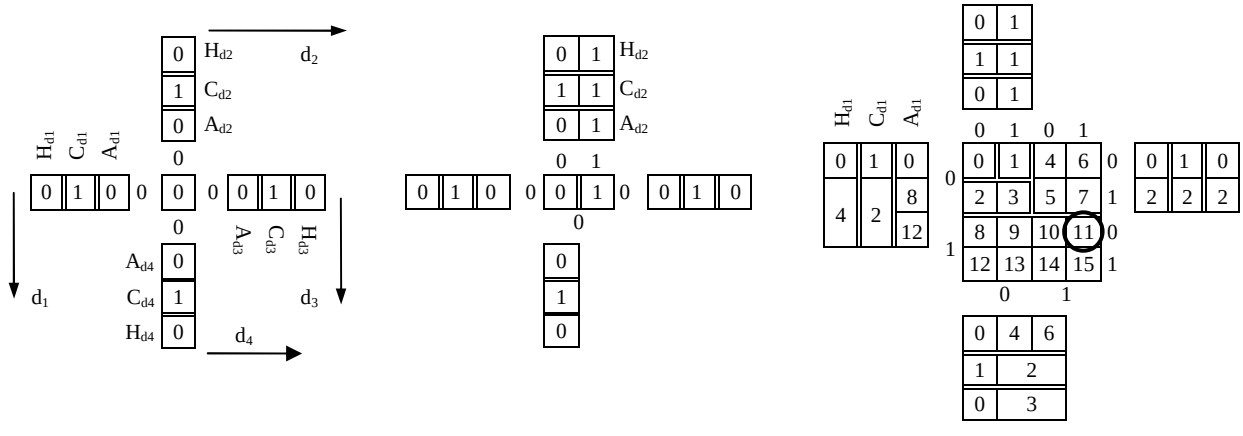
Consider a 4-dimensional array of size A[l_1, l_2, l_3, l_4] where l_i (i = 1, 2, 3, 4) indicates the length of each dimension d_i that varies from 0 to $l_i - 1$. The dimension (d_1, d_3) and (d_2, d_4) are grouped as adjacent dimensions respectively. The length of the extended subarray which is allocated dynamically for the extension along dimension d_1(say), is determined by $l_2 \times l_3 \times l_4$ (i.e. other 3 dimensions). The number of segments in the subarray is the length of the adjacent dimension, $adj(d_1) = d_3$. Therefore, the size of each segment of a subarray extended along dimension d_1 is determined by $l_2 \times l_4$. After extending along dimension d_1, the length of the corresponding dimension is incremented by 1. For each extension the auxiliary tables namely history table, address table and coefficient tables are maintained.

For each history, the address table contains the first address of the extended subarrays for the corresponding dimension. For each extension the subarrays are broken into segments. The address table stores the first addresses of each segments of the subarray. Hence for a single subarray (i.e. history value) the address table can have more than one entry. History table contains the construction history of the subarrays. There is a history counter that counts the construction history of the subarrays. Coefficient table contains the coefficient of subarrays. 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. Fig. 2 shows the details extension realization of a 4 dimensional EKA. It also displays how the different auxiliary tables are maintained during the extension along a particular dimension. Fig. 2(a) shows the initial setup with history counter 0 stored in history tables, address tables point to the first address of the physical array, and coefficients tables entry is 1, since length of each dimension is 1. During extension along d_1 or d_3 the segment size is $l_2 \times l_4$, so we chose l_2 as coefficient vector. Similarly, l_3 is used as coefficient vector for extension along d_3 or d_4. Fig. 2(b) shows the extension along d_2 dimension, the incremented history value 1 is stored in history table of dimension 2, H_{d2}. Since l_3 is 1 C_{d2} stores this value and address table points to the first address which is 1. Fig 2(c) shows the extension of d_1 dimension considering that Fig. 2(b) is already extended once in d_3, and d_4 dimension. As it is already extended in d_3, and d_4 dimension, the history value reaches to 3, now for extending in d_1 the value becomes 4 which is stored in H_{d1}. Coefficient table entry is 2 because of the l_2 is 2. The size of each segment is $l_2 \times l_4 = 2 \times 2 = 4$, and the number of segments depends on length of $adj(d_1)$, that is l_3, which is 2 here. Therefore the address table will have two entry pointed to the first address of each segment which is shown in the Fig. 2(c).

B. n Dimensional EKA

The EKA scheme can be generalized to *n* dimensions using a set of EKA(4)s. Fig. 3 shows an EKA(6) represented by a set of EKA(4) in two level having 5th and 6th dimension of lengths 3 and 2 respectively. 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



(a) Initial setup (b) Extension along d_2 dimension (c) Extension along d_1 dimension

Figure 2. Extension realization of EKA(4).

using the subscripts of higher dimensions. For the case of EKA(*n*), similar hierarchical structure will be needed. to locate the appropriate EKA(4). Hence the EKA(*n*) is a set of EKA(4)s and a set of pointer arrays. When extension is necessary on a dimension ≤ 4, all the EKA(4)s are extended.

Since an *n* dimensional array is represented by a set of 4 dimensional arrays, the extension subarrays are always 3-dimensional and that are again broken into 2-dimensional segments. When number of dimension is large (greater than 4), then one dimensional pointer arrays are incorporated. Therefore occurrence of address space or consecutive memory space overflow will be delayed even for very large values of number of dimensions or length of dimension.

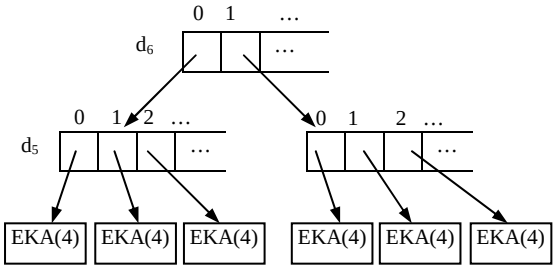


Figure 3. Realization of 6 dimensional EKA

C. Retrieval in EKA(4)

Let the value to be retrieved is indicated by the subscript (x_1, x_2, x_3, x_4). 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_1) that corresponds to history value h_{max} is determined. h_{max} is the subarray that contains our desired element. Now the first address and offset from the first address is to be found out. The adjacent dimension *adj*(d_1) (say d_3) and its subscript x_3 is found. The first address is found from $H_{d1}[x_1].A_{d1}[x_3]$. The *offset* from the first address is computed using the addressing function; the coefficient vectors are stored in C_{d1}. Then adding the *offset* with the first address, the desired array cell (x_1, x_2, x_3, x_4) is found.

Example 1**:** Let four subscripts (1, 1, 0, 1) for dimension d_1, d_2, d_3, and d_4 is given (See Fig. 2(c)). Here h_{max} = max($H_{d1}[1] = 4$, $H_{d2}[1] = 1$, $H_{d3}[0] = 0$, $H_{d4}[1] = 3$) =4, and dimension corresponding to h_{max} is d_1 whose subscript is 1 and *adj*(d_1) = d_3 and x_3 = 0. So the first address is in $A_{d1}[1][0] = 8$, and offset is calculated using the coefficient vector stored in coefficient table C_{d1} which is 2. Here offset = $C_{d1}[2] * x_4 + x_2 = 2*1+1 = 3$. Finally adding the first address with the offset the desired location 8 + 3 = 11 is found (encircled in Fig. 2(c)).

D. Retrieval in 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) are the set one dimensional pointer arrays that points to next lower dimensions. Hence using the subscripts x_k (d_k > 4) the pointer arrays are searched to locate the lower dimensions (See Fig. 3) until we find the The set of EKA(4)s stores the actual data values and the hierarchical arrays are simply served as indexes and used desired EKA(4). After that, using the above computation technique the location in EKA(4) can be found.

IV. History Segment-Offset Encoding on EKA

Generally an element of an *n*-dimensional array is accessed by a subscript of *n* indices. But we are here going to present an encoding scheme called as History Segment-Offset Encoding (HSOE) based on EKA. This scheme uses only three value *History value*, *Segment number*, and *Offset* within a segment to access the desired array cell.

A. Realization of HSOE on EKA(4)

In addition to the auxiliary tables of EKA mentioned in section III, there needs an additional auxiliary table namely *Element* table 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 Fig. 4(a) which is actually the real array after extending the array of Fig. 2(c) in d_3 and d_2 dimension respectively. 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 segment offset encoded representation of the array is shown in Fig. 4(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 $E_{d2}[2] = 4$, because subarray 6 has two segments, and the last segment has 4 elements. In the centre of Fig. 4(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 HSOE 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 III C. 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

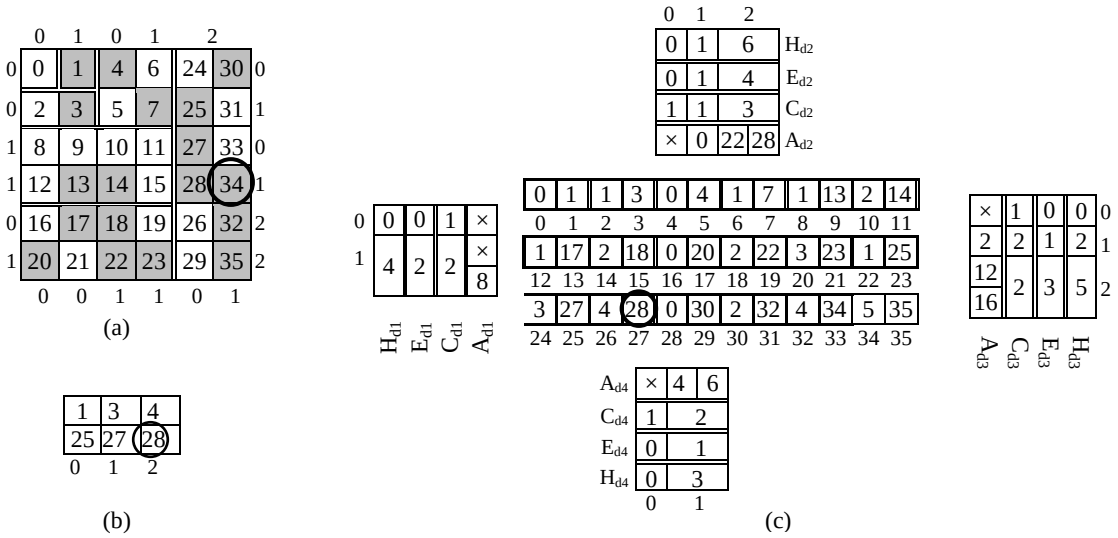


Figure 4. History Segment-Offset representation of EKA(4).

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.

Example: Let four subscripts ⟨1, 2, 1, 0⟩ for dimension d_1, d_2, d_3, and d_4 is given (see Fig. 4). Here h_{max} = max($H_{d1}[1]$, $H_{d2}[2]$, $H_{d3}[1]$, $H_{d4}[0]$)= max(4, 6, 2, 0) =6, and dimension corresponding to h_{max} i.e. $d_{max} = d_2$ whose subscript $x_{max} = 2$ and $adj(d_{max}) = adj(d_2) = d_4 = d_{adj}$ and $x_{adj} = 0$. So the *firstAddress* = $A_{d2}[2][0] = 22$, and offset is calculated using the coefficient vector stored in coefficient table C_{d2} which is 3. Here, *offset* = $C_{d2}[2] * x_3 + x_1 = 3*1 + 1 = 4$. Now the segment is loaded into memory (Fig. 4(b)), and binary search finds the offset 4, therefore the desired value is 28 (encircled in Fig. 4(b), (c)).

Backward Mapping on HSOE EKA(4): Let we are given ⟨*h*, *s*, *o*⟩ that represents history value, the segment number, and an offset position respectively in a HSOE 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* i.e. $C_{di}[x] = c$, then two other dimensional subscripts x_u, x_v (say) can be found by the formula: $x_u = offset \% c$; $x_v = offset \backslash c$, where % is a 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 Fig. 4(a)).

B. Realization of HSOE 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 HSOE over each EKA(4)s on a iterative manner. Forward mapping described above (section IV) 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 *uppersubscripts* array of length two. 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 $4ln^{-3}$. 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), means

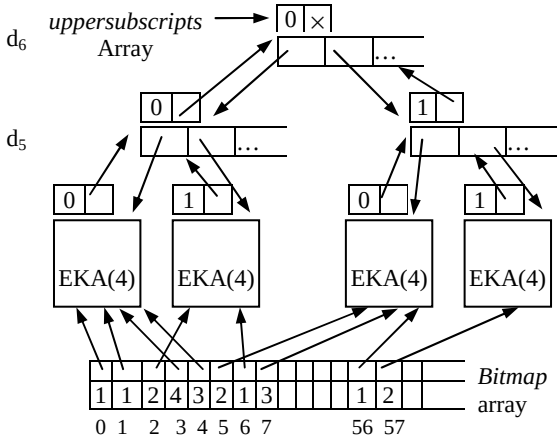


Figure 5. Arrangement of HSOE EKA(*n*) for backward mapping.

the dimension of extension and another is a pointer to the EKA(4). Fig. 5 shows the logical arrangement of an HSOE EKA(*n*) along with necessary auxiliary tables required for backward mapping.

Backward Mapping on HSOE EKA(n): Let, given values are ⟨*h,s,o*⟩. 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 IV.A. Since each EKA(4) maintains a *upper subscripts* table, the higher dimensional subscripts can be found from there by going back to root and by collecting *upper subscripts* array entry.

V. Performance Results

We have constructed the TMA and EKA systems placing the array in the secondary storage having the parameter values shown in Table I. 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 4KB using Microsoft visual C++ compiler. We will show that without any retrieval penalty, the overall scope of the array can be extended in terms of memory allocation and address space allocation of a multidimensional array effectively if implemented using EKA. We used the parameters *n* and *l* to represent number of dimension and length of dimension respectively for both TMA and EKA. λ represents the length of extension in each dimension. For simplicity we extend the dimensions of TMA and EKA in round robin fashion.

A. Retrieval Cost

Fig 6(a) shows the retrieval times for range key query on EKA and TMA for $n = 5$ with different known dimension. The retrieval cost is dependent on the known dimension along which the range for retrieval is done. The average of the retrieval cost for n=5 is shown in 6(c). We see that the average retrieval time is nearly same for EKA and TMA. Fig. 6(b) and 6(d) shows average retrieval cost for $n = 4$ and 6 respectively. The results show that the retrieval cost is similar for both TMA and EKA and it can be concluded that there is no retrieval penalty for EKA over TMA.

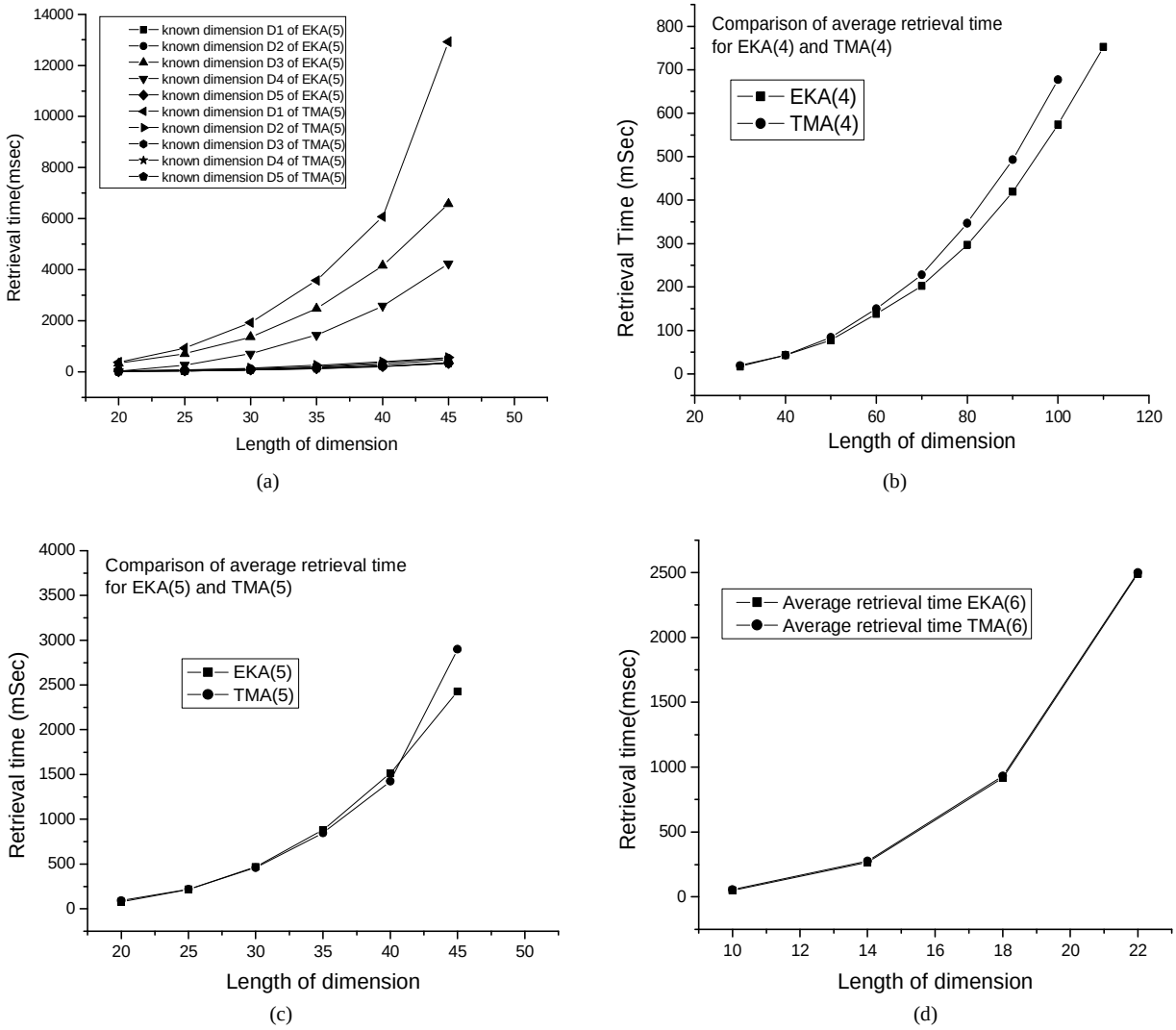
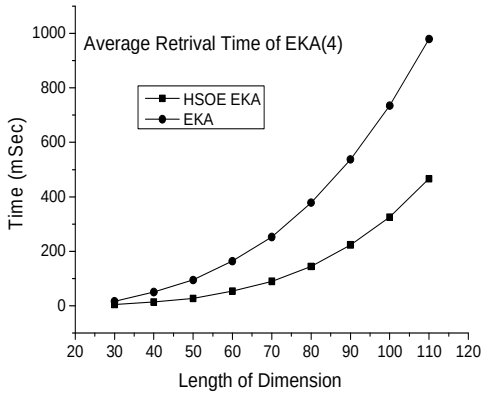


Figure 6. Comparison of retrieval times of EKA and TMA.

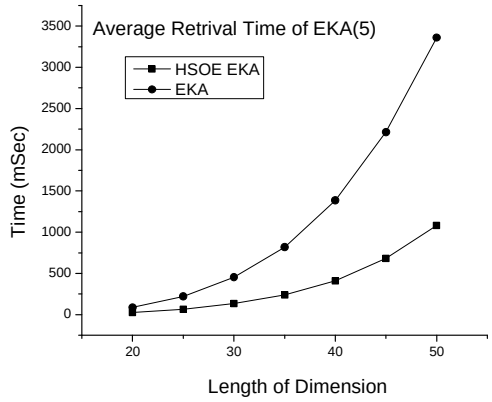
TABLE I.

ASSUMED PARAMETERS FOR CONSTRUCTED PROTOTYPES

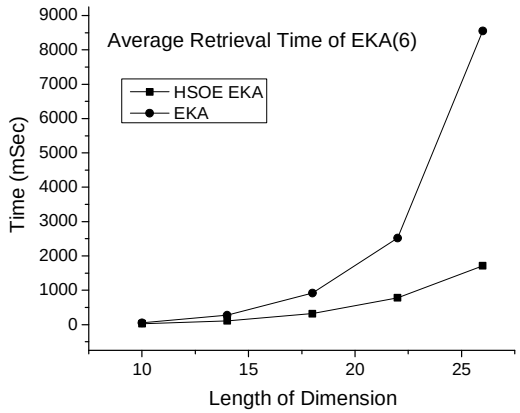
n	λ	max(l_i)	Initial V = l^n	NRQ Subscripts
4	10	100	$(30)^4$	(l–λ)/2 to (l+λ)/2
5	5	45	$(20)^5$	
6	2	22	$(10)^6$	



(a)



(b)



(c)

Figure 7. Average retrieval time comparison between EKA and HSOE EKA

Fig. 7(a), 7(b), and 7(c) show the comparison of range key retrieval times of NRQ subscripts in straight EKA and HSOE EKA for number of dimensions n = 4, 5, 6 respectively. HSOE EKA is practically suitable for representing sparse array where frequency of range query is high. Here given retrieval time is the average of retrieval times with array density ρ = 0.4, 0.5, and 0.6. However the retrieval time with a particular density is, in fact, an average retrieval time considering each dimension as known dimension. In every case the HSOE EKA needs much less time than straight or pure EKA representation, which is depicted in Fig. 7. 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 EKA whatever the density factor 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 HSOW 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 HSOE EKA is better than straight EKA.

B. Overflow

In multidimensional array, the location of an element is calculated using the addressing function described in Section III. 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, maximum value will be l^2, which greatly delays the overflow.

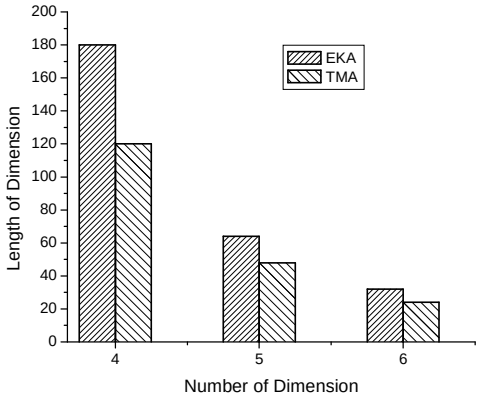
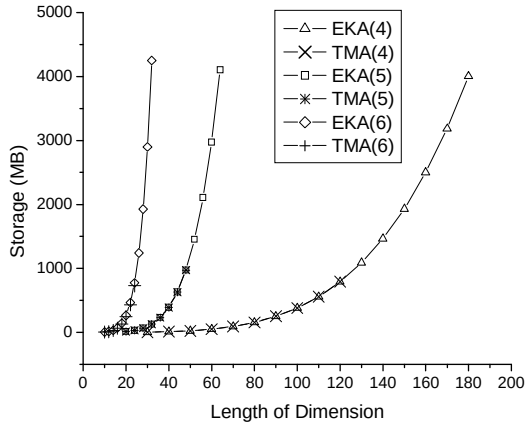


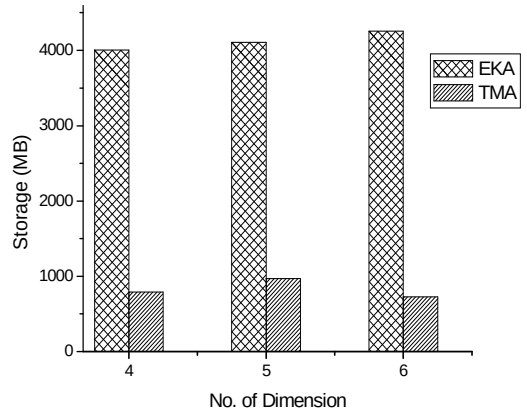
Figure 8. Maximum length in each dimension before overflow.

Fig. 8 shows the maximum length of dimension that is reached by the EKA and TMA before the occurrence of overflow the memory space for varying number of dimensions. From Fig. 8, 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.



(a) Total storage requirement.



(b) Maximum storage allocated before the occurrence of overflow

Figure 9. Storage allocation of EKA and TMA.

Fig. 9(a) shows the total storage requirement for EKA and TMA on different number of dimensions varying the length of dimension. From Fig. 9(a), it is found that both EKA and TMA need almost same amount of storage up to a particular length of dimension. So we can conclude that the nature of storage requirement is almost same for EKA and TMA. Fig. 9(b) shows the maximum storage allocated for EKA and TMA on different number of dimension before reaching to overflow situation. From Fig. 9(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.

C. Space complexity

The number of cells in a history or coefficient or element table for any dimension is *l*, where *l* is the length of a dimension. Since for each dimension these tables exist (See Section IV.A) and if we consider α bytes for each cell then size of these three table becomes $3\times4\times l\times\alpha = 12\alpha l$. Similarly we can find that size of address table is $\alpha(2l^2 + 2)$. Let the density of data of the array is ρ, and each offset of the nonempty cell and the value of nonempty cells are represented by α and β bytes respectively. Therefore the size of the HSOE EKA(4) is $\alpha(\rho l^4 + 2l^2 + 12l + 2) + \beta\rho l^4$. HSOE EKA($n \geq 5$) has some higher dimensional pointers and each pointed to an HSOE EKA(4), so it can be shown that space needed for this case is $\alpha(\rho l^n + 2l^{n-2} + 12l^{n-3} + 3l^{n-4}) + \beta\rho l^n$. Hence space complexity is $O(l^n)$. The space complexity for CRS/CCS and ECRS/ECCS scheme [24] is also $O(l^n)$.

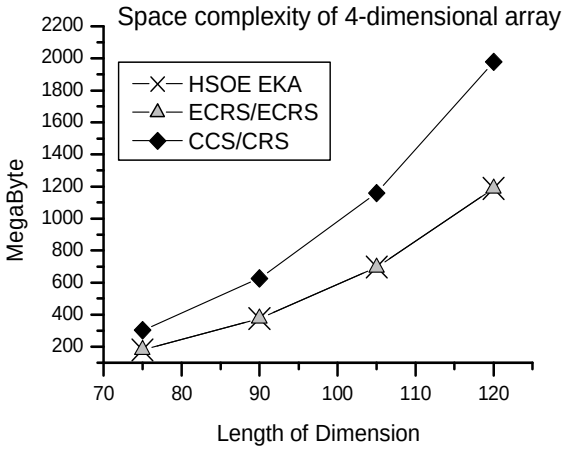


Figure 10. Space requirement for HSOE, ECRS/ECCS and CRS/CCS

Fig. 10 shows the space requirement of different scheme for a 4-dimensional array with $\rho = 0.5$. HSOE requires same amount of space as required for ECRS/ECCS[24] and less amount CRS/CCS for varying length of dimension. Moreover HSOE scheme is dynamically extendible but ECRS/ECCS and CRS/CCS do not have this property. Hence we conclude that HSOE scheme outperforms ECRS/ECCS and CRS/CCS.

VI. CONCLUSION

In this paper, we proposed and evaluated a new implementation scheme based on 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. Most of the array representation systems do not consider the address space overflow problem which is consider here. This scheme can be successfully applied to database applications especially for multidimensional database or multidimensional data warehousing system. This is because representing a relational table as multidimensional array suitable for various aggregations but it creates a problem of high degree of sparsity. We have shown that our encoding scheme improves the retrieval performance of the sparse array. 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] K. E. Seamons and M. Winslett, "Physical schemas for large multidimensional arrays in scientific computing applications," *Proc. Of SSDBM*, pp. 218–227, 1994.

[2] S. Sarawagi and M. Stonebraker, "Efficient organization of large multidimensional arrays," *Proc. of ICDE*, pp. 328–336, 1994.

[3] Y. Zhao, P. M. Deshpande, and J. F. Naughton, "An array based algorithm for simultaneous multidimensional aggregates," *ACM SIGMOD*, 159–170, 1997.

[4] E. J. Otoo and T. H. Merrett, "A storage scheme for extendible arrays", *Computing*, Vol. 31, pp. 1-9, 1983.

[5] D. Rotem and J. L. Zhao, "Extendible arrays for statistical databases and OLAP applications," *Proc. of Scientific and Statistical Database Management*, pp. 108–117, 1996.

[6] K. M. A. Hasan, M. Kuroda, N. Azuma, T. Tsuji, and K. Higuchi, "An extendible array based implementation of relational tables for multidimensional databases," *Proc. of DaWak, LNCS*, pp. 233–242, 2005.

[7] T. Tsuji, M. Kuroda, and K. Higuchi, "History offset implementation scheme for large scale multidimensional data sets," *Proc. of ACM Symposium on Applied Computing*, pp. 1021–1028, 2008.

[8] K. M. A. Hasan, T. Tsuji, and K. Higuchi, "An efficient implementation for MOLAP basic data structure and its evaluation," *Proc. of DASFAA, LNCS 4443*, pp. 288–299, 2007.

[9] S. M. M. Ahsan and K. M. A. Hasan, "An implementation scheme for multidimensional extendable array operations and its evaluation," *Proc. of the International Conference on Informatics Engineering & Information Science, CCIS 253, Springer, Heidelberg*, pp. 136–150, 2011.

[10] S. Sidiroglou, G. Giovanidis, and A. D. Keromytis, "A dynamic mechanism for recovering from buffer overflow attacks," *Information Security Conference/Information Security Workshop - ISC(ISW)*, pp. 1–15, 2005.

[11] T. Chiueh and F. Hsu, "RAD: a compile-time solution to buffer overflow attacks," *Proc. of ICDCS*, pp. 409–417, 2001.

[12] T. B. Pedersen and C. S. Jensen, "Multidimensional database technology," *IEEE Computer*, 34(12), pp. 40–46, 2001.

[13] L. Chun, C. C. Yeh, and S. L. Jen, "Efficient representation scheme for multidimensional array operations," *IEEE Computer*, 51(3), pp. 327–345, 2002.

[14] Y. L. Chun, C. C. Yeh, and S. L. Jen, "Efficient data parallel algorithms for multidimensional array operations based on the EKMR scheme for distributed memory multicomputer," *IEEE Parallel and Distributed Systems*, 14(7), pp. 625–639, 2003.

[15] E. J. Otoo and D. Rotem, "Efficient storage allocation of large-scale extendible multidimensional scientific datasets," *Proc. of the 18th International Conference on Scientific and Statistical Database Management*, Vienna, Austria, pp. 179–183, 2006.

[16] D. Rotem, E. J. Otoo, and S. Seshadri, *Chunking of Large Multidimensional Arrays*," Lawrence Berkeley National Laboratory, University of California, University of California, LBNL-63230, 2007.

[17] K. M. A. Hasan, T. Tsuji, and K. Higuchi, "A range key query scheme for multidimensional databases", *Proc. of the 5th ICECE*, pp. 958–963, 2008.

[18] K. M. A. Hasan, T. Tsuji, K. Higuchi, "A parallel implementation scheme of relational tables based on multidimensional Extendible array", *Int. Journal of Data Warehousing and Mining*, 2(4). pp.66–85, 2006.

[19] Y. Shao, P. M. Deshpande, and J. f. Naughton, "An Array Based Algorithm for Simultaneous Multidimensional Aggregate," Proceedings of SIGMOD'97, pp. 159–170, 1997.

[20] T. Tsuji, A. Hara, and K. Higuchi, "An extendible multidimensional array system for MOLAP," *SAC'06*, Dijon, France, pp. 23─27, 2006.

[21] S. M. M. Ahsan and K. M. A. Hasan "A solution of address space overflow for large multidimensional arrays," *Proc. of 14th ICCIT*, pp. 381–386, 2011.

[22] K. M. A. Hasan, "Compression schemes of high dimensional data for MOLAP", *Evolving Application Domains of Data Warehousing and Mining: Trends and Solutions*, pp. 64-81, 2009.

[23] J. B. White and P. Sadayappan, "On improving the performance of sparse matrixvector multiplication", *Proc. of International Conference on High Performance Computing*, pp. 711─725, 1997.

[24] Y. L. Chun, C. C. Yeh, and S. L. Jen, "Efficient data compression methods for multidimensional sparse array operations based on the EKMR scheme," *IEEE Computer*, Vol. 52, No. 12, pp. 1640–1646, 2003.

[25] M. M. Mano, *Digitial Logic and Computer Design*, Prentice Hall, 2005.

Sk. Md. Masudul Ahsan, born in December, 1980 in Narsingdi, Bangladesh. He received his B.Sc. and M.Sc. degrees in computer science & engineering from Khulna University of Engineering & Technology, Bangladesh in 2003 and 2012 respectively. He is now serving as an assistant professor in the Department of Computer Science and Engineering at Khulna University of Engineering & Technology. His current research interests include database implementation schemes, data warehousing system, visual modeling, and machine vision.

K. M. Azharul Hasan received his B.Sc. (Engg.) from Khulna University, Bangladesh in 1999 and M. E. from Asian Institute of Technology (AIT), Thailand in 2002 both in Computer Science. He received his Ph.D. from the Graduate School of Engineering, University of Fukui, Japan in 2006. His research interest lies in the areas of databases and his main research interests include Data warehousing, MOLAP, Multidimensional databases, Parallel and distributed databases, Parallel algorithms, Information retrieval, Software metric and Software maintenance. He is with the Department of Computer Science and Engineering Khulna University of Engineering and Technology (KUET), Bangladesh since 2001.