Journal of Computing and Information Technology - CIT 23, 2015, 2, 111–121
doi:10.2498/cit.1002441

Segment Oriented Compression Scheme for MOLAP Based on Extendible Multidimensional Arrays

Sk. Md. Masudul Ahsan and K. M. Azharul Hasan

Department of Computer Science and Engineering, Khulna University of Engineering and Technology (KUET), Bangladesh

Many statistical and MOLAP applications use multidimensional arrays as the basic data structure to allow the efficient and convenient storage and retrieval of large volumes of business data for decision making. Allocation of data or data compression is a key performance factor for this purpose because performance strongly depends on the amount of storage required and availability of memory. This holds especially for data warehousing environments in which huge amounts of data have to be dealt with. The most evident consequence of data compression is that it reduces storage cost by packing more logical data per unit of physical capacity. And improved performance is a net outcome because less physical data need to be retrieved during scan-oriented queries. In this paper, an efficient data compression technique is proposed based on the notion of extendible array. The main idea of the scheme is to compress each of the segments of the extendible array using the position information only. We compare the proposed scheme for different performance issues with prominent compression schemes.

Keywords: extendible array, multidimensional array, MOLAP, database compression, compression ratio

1. Introduction

The strong need to handle large scale data efficiently has been promoting comprehensive research themes on the organization or implementation schemes for multidimensional arrays [1]. Multidimensional arrays are the basic data structure used in many scientific, statistical, and engineering applications because they are well understood and can easily be incorporated in other data structures [2]. Scientific and statistical datasets grow too massively in their size in the order of terabytes and petabytes. Hence the storage of such data requires efficient dynamic storage schemes where the array is allowed to arbitrarily extend the bounds of the dimensions [3], [4]. Conventional multidimensional arrays do not support dynamic extension of an array and hence the addition of a new column value is impossible if the size of the dimension overflows. Therefore, we need a method of extending multidimensional arrays in all dimensions [5], [6]. Another problem with the multidimensional array structure is its sparsity, which wastes memory because a large number of array cells are empty and thus are rarely used for actual implementation [7], [8]. In particular, the sparsity problem becomes serious 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 stored would not increase at such a rate. So it is important to design a technique to compress such sparse arrays.

In our previous work [9], [10], we present a new extendible multidimensional array system, namely Extendible Karnaugh Array (EKA)along with its operations and [3] presents memory management performance for controlling the address space overflow for large multidimensional arrays. The EKA is based on Karnaugh map [11] and is dynamically extendible during runtime in any direction without any relocation of the data already stored. In this paper, we propose a compression scheme based on the EKA [9], [10]. To compress the EKA, we store only the position information of each segment of the array, i.e. the construction history, the segment number and the offset inside the array. Our scheme will be called SCEKA (Segment based Compression scheme for Extended Karnaugh Array). It can be effectively applied not only to the implementation of MOLAP [7], but also to multidimensional database systems [12], or parallel data warehouse systems [13], [14].

The rest of the paper is organized as follows; Section 2 gives a brief description of the EKA system, Section 3 describes the proposed compression scheme, Section 4 explains the experimental results, Section 5 provides some comparison with other related works and finally Section 6 outlines the conclusion.

2. The EKA System

The idea of EKA [9], [10] is based on Karnaugh Map (K-map) which is used for minimizing Boolean expressions. Figure 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 that indicates the possible combinations in a two dimensional array for the four Boolean variables. The array representation of the K-map for 4 variable Boolean function is shown in Figure 1(b).

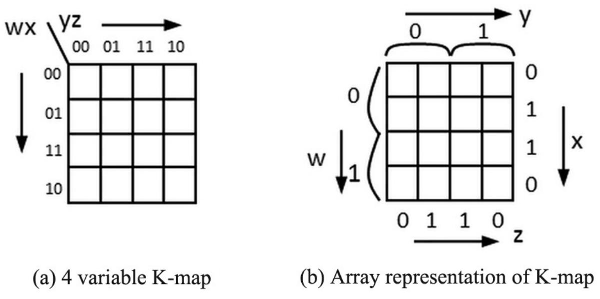


Figure 1. Realization of Boolean function using K-map.

Definition 1: (Adjacent Dimension). *The dimensions (or index variables) that are placed together in the Boolean function representation of K-map are termed adjacent dimensions* (written *adj*$(i) = j$). The dimensions (w, x) are the adjacent dimensions in Figure 1(a) and (b) i.e. *adj*$(w) = x$ or *adj*$(x) = w$.

Figure 2 shows the dynamic extension realization of the array of Figure 1(b). EKA is an

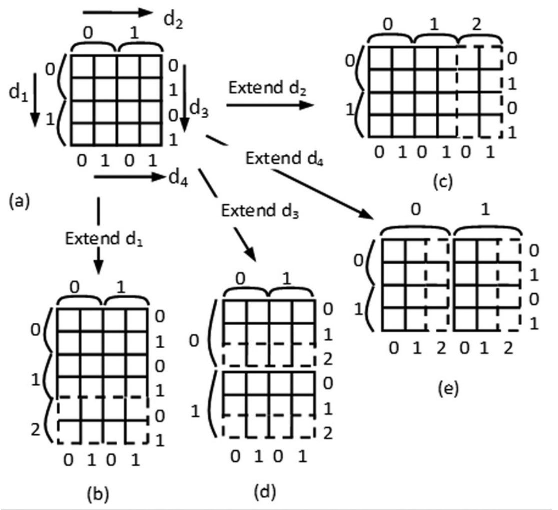


Figure 2. Logical extension of 4-dimensional EKA.

array system which is the combination of sub arrays. To maintain dynamic extension and the subarrays, it has three types of auxiliary tables, namely *history table*, *coefficient table*, and *address table*. For each dimension these tables exist. The *history table* stores construction history of the subarrays. The first address of a segment is stored in the *address table* and is used to compute the correct position of an element. Any element in the n dimensional array is determined by an addressing function as follows

$$
\begin{aligned}
& f(x_n, x_{n-1}, x_{n-2}, \ldots, x_2, x_1) \\
&\quad = l_1 l_2 \ldots l_{n-1} x_n + l_1 l_2 \ldots l_{n-2} x_{n-1} \\
&\qquad + \ldots + l_1 x_2 + x_1
\end{aligned}
$$

The coefficients of the addressing function, namely $\langle l_1 l_2 \ldots l_{n-1}, l_1 l_2 \ldots l_{n-2}, \ldots, l_1 \rangle$, are referred to as coefficient vectors which are stored in *coefficient table*. The subarrays are divided into equal size segments (see Figure 2) that can be stored contiguously on the disk.

Consider a 4-dimensional array of size A$[l_1, l_2, l_3, l_4]$ where l_i $(i = 1, 2, 3, 4)$ is the length of each dimension d_i that varies from 0 to $l_i - 1$. The dimensions $(\mathrm{d}_1, \mathrm{d}_3)$ and $(\mathrm{d}_2, \mathrm{d}_4)$ are grouped as adjacent dimensions respectively. The length of the extended subarray which is allocated dynamically for the extension along dimension d_i is determined by $\prod_{j=1}^{4} \mathrm{d}_j$ $(j \neq i)$.

The number of segments in any subarray (belongs to dimension d_i) is determined by the

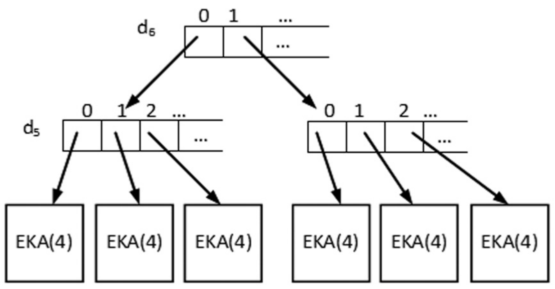


Figure 3. The realization of EKA(n).

length of the adjacent dimension of d_i. The number of segments determines the number of entries in the address table and is equal to the length of adjacent dimension. After extending along any dimension d_i, the length of the corresponding dimension is incremented by 1. For each extension of the auxiliary tables, namely *history table*, *address table* and *coefficient tables* are maintained. Figure 4 shows the detailed extension realization of a 4 dimensional EKA where H_{d_1}, C_{d_1} and A_{d_1} are *history table*, *coefficient table* and *address tables* of dimension 1. The EKA scheme can be generalized to n dimensions using a set of EKA(4)s. We write EKA(n) to denote an n dimensional EKA. Figure 3 shows an EKA(6) represented by a set of EKA(4)s on two levels containing 5th and 6th dimensions each of lengths 3 and 2 respectively. Each of 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. In the case of EKA(n), the 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.

The segments are always 2 dimensional for an EKA(n). Hence, in our model the coefficient vector becomes single dimensional such as $\langle l_1 \rangle$ only. The EKA can be extended along any dimension dynamically during runtime only by the cost of these auxiliary tables.

Let the value stored in the subscript (x_1, x_2, x_3, x_4) be retrieved. The maximum history value among the subscripts $h_{\max} = \max(H_{d_1}[x_1], H_{d_2}[x_2], H_{d_3}[x_3], H_{d_4}[x_4])$ and its corresponding dimension (say d_1) that corresponds to the $h_{\max}$ is determined. $h_{\max}$ is the subarray that contains our desired element. The first address and offset from the first address is found out using the auxiliary tables. The adjacent dimension $adj(d_1)$ (say d_3) and its subscript x_3 is found. The first address is found from $H_{d_1}[x_1].A_{d_1}[x_3]$. The *offset* from the first address is computed using the addressing function; coefficient vectors are stored in C_{d_1}. Then adding the *offset* with the first address, the desired array cell (x_1, x_2, x_3, x_4) is found.

For example, the values of the subscripts $(2, 2, 0, 0)$ are determined as follows (see Figure 4). Here $h_{\max} = \max(H_{d_1}[2] = 7,\ H_{d_2}[2] = 6,\ H_{d_3}[0] = 0,\ H_{d_4}[0] = 0) = 7$, and dimension corresponding to $h_{\max}$ is d_1 whose subscript is $x_1 = 2$ and $adj(d_1) = d_3$ and $x_3 = 0$. So the first address is in $H_{d_1}[2].A_{d_1}[0] = 36$, and offset is calculated using the coefficient vector stored in coefficient table $C_{d_1}[2] = 3$. Here offset $= C_{d_1}[2] * x_4 + x_2 = 3 * 0 + 2 = 2$. Finally, adding the first address with the offset, the desired location $36 + 2 = 38$ is found (encircled in Figure 4).

3. The SCEKA

The Segment based Compression scheme for Extended Karnaugh Array (SCEKA) stores the

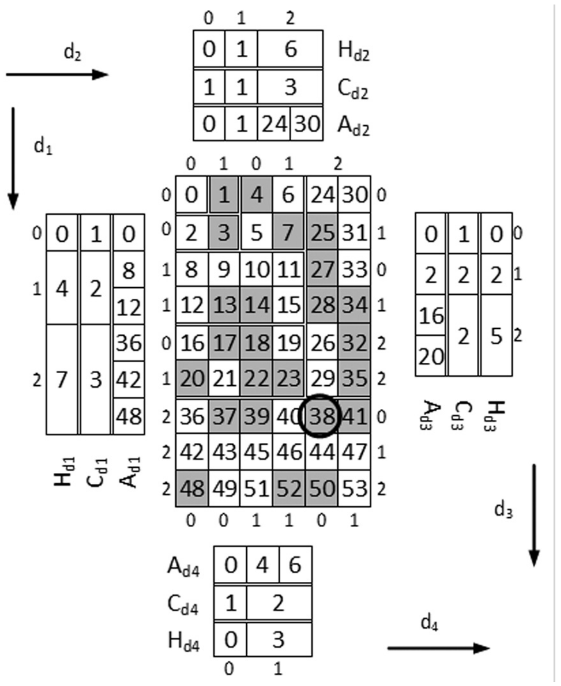


Figure 4. Segment-compressed representation of EKA(4).

extended history value, the segment number and the logical location inside the segment of the subarray i.e. the offset of the subarray. The data stored in the SCEKA scheme can be accessed in compressed form and at the same time it can grow and shrink in length or number of dimensions at run time.

3.1. The Realization of SCEKA(4)

Along with the auxiliary tables of EKA, SCEKA uses an additional table namely *Element* table for each dimension to store the number of elements in a segment. In SCEKA scheme, we store the auxiliary tables including the element table. Along with the auxiliary tables, we also store the segment number of the subarray to which the element belongs and the offset of element of the segment. These are the position information to recollect the subscripts of the item. Hence, in SCEKA scheme, we store the tuple $\langle$ *history value*, *segment number*, *offset*$\rangle$ for array cell mapping and the data is stored as well. The history value is unique and can uniquely determine the subarray. The segment number inside the subarray is also unique and can also be determined uniquely. The offset value inside the segment is also unique and can be determined by the addressing function. Hence the tuple $\langle$ *history value*, *segment number*, *offset*$\rangle$ can uniquely map anarray cell of the EKA. Let the subscripts $\langle x_1, x_2, x_3, x_4\rangle$ are given; we determine the $\langle$ *history value*, *segment number*, *offset*$\rangle$ as follows: To find the history value, we calculate $h_{\max}$ as described in Section 2. To determine the segment number we calculate the dimension of $\mathrm{h}_{\max}$ as $\mathrm{d}_{\max}$ and hence the $adj(\mathrm{d}_{\max}) = \mathrm{d}_k$ $(k = 1 \ldots 4)$ is found out. Hence x_k is found and this x_k is the segment number s. Once the segment number s is found, the first address of the segment can be found from the address table and the offset is calculated. We store the offset and data value pair in a linear array in the secondary storage as compressed physical array.

Example 1. Let us be given four subscripts $\langle 2, 2, 0, 0\rangle$ for dimension d_1, d_2, d_3, and d_4 (see Figure 4). Here $h_{\max} = \max(\mathrm{H}_{\mathrm{d}_1}[2], \mathrm{H}_{\mathrm{d}_2}[2], \mathrm{H}_{\mathrm{d}_3}[0], \mathrm{H}_{\mathrm{d}_4}[1]) = \max(7, 6, 0, 0) = 7$, and dimension corresponding to $h_{\max}$ i.e. $\mathrm{d}_{\max} = \mathrm{d}_1$ whose subscript $x_{\max} = 2$ and $adj(\mathrm{d}_{\max}) = adj(\mathrm{d}_1) = \mathrm{d}_3$ and $x_3 = 0$ (i.e. segment number).

So the *firstAddress* $= A_{\mathrm{d}_1}[2][0] = 36$, and offset is calculated using the coefficient vector stored in coefficient table $\mathrm{C}_{\mathrm{d}_1}$ which is 3. Here, *offset* $= \mathrm{C}_{\mathrm{d}_1}[2]{*}x_4 + x_2 = 3{*}0 + 2 = 2$. Hence, to map the cell the tuple $\langle 7, 0, 2\rangle$ is stored along with the data value 38 (encircled in Figure 4).

3.2. Accessing from SCEKA(4)

Let us be given the SCEKA tuple $\langle h, s, o\rangle$ 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 searching the history tables with the given h we find the dimension i $(\mathrm{H}_{\mathrm{d}_i})$ and its subscript x_i. Let $adj(\mathrm{d}_i) = \mathrm{d}_j$, then x_j is the segment number s i.e. $(x_j = s)$. The other two subscripts x_u, x_v (say) are found using the entry in co-efficient table as follows. Let the coefficient table entry in dimension d_i at x_i is c i.e. $\mathrm{C}_{\mathrm{d}_i}[x_i] = \mathrm{c}$, then

$$x_u = \text{offset} \,\%\, \text{c} \quad \text{and} \quad x_v = \text{offset} \backslash \text{c}$$

where % is a modulus and \ is an integer division operator.

Example 2: Let the tuple $\langle 6, 1, 4\rangle$ be given, that is, history = 6, segment number = 1, offset = 4. Now searching each of the history tables, we found that $\mathrm{H}_{\mathrm{d}_2}[2] = 6$, so $x_2 = 2$. Here $adj(\mathrm{d}_2) = \mathrm{d}_4$, so $x_4 = 1$ (the segment number). Again, we see that $\mathrm{C}_{\mathrm{d}_2}[2] = 3 = c$, which was the length of dimension 3 during the extension. So $x_3 = \text{offset} \,\%\, 3 = 4 \,\%\, 3 = 1$, and $x_1 = \text{offset}\backslash 3 = 4\backslash 3 = 1$. Hence the desired subscript is $\langle 1, 2, 1, 1\rangle$. If the *first Address* is null, the element doesn't exist.

3.3. Realization of SCEKA(n)

We only compress each of the EKA(4)s and the upper pointer arrays remain as usual. Since an n-dimensional EKA can be represented as a collection of EKA(4)s, we can individually compress each EKA(4)s. The compression scheme

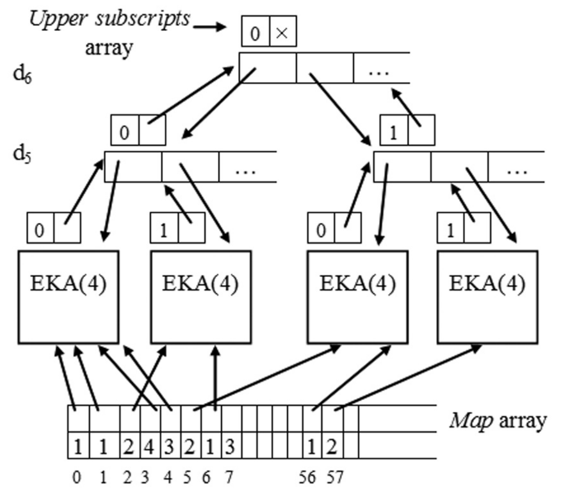


Figure 5. Arrangement of SCEKA(n) for backward mapping.

described above is applied to each of those EKA(4)s. To access a compressed SCEKA(n), we need some additional tables, since the EKA scheme loses the higher dimensional subscripts to search in bottom up manner. So, each EKA(4) and higher dimensional pointer arrays will maintain a tiny *Upper subscripts* array (see Figure 5) to get the facility for searching the bottom up fashion. It will contain the index of immediate lower and higher dimension pointers.

To reduce the search space, we employ a small two dimensional array namely *map* array. The index of the array represents the history counter (history values), one of its entry is the dimension of extension and another is a pointer to the EKA(4). Consider the tuple $\langle h, s, o \rangle$. We first look at the *map* array at index h and find the entry i(say) which is the dimension of h and the exact EKA(4) in which dimension the h resides. Now we apply binary search only over the history table of dimension i, $\mathrm{H}_{\mathrm{d}_i}$ to locate the position of h. Now we can determine the subscripts of the lowest 4 dimensions by applying the process described in Section 3. Since each EKA(4) maintains an *upper subscripts* table, the higher dimensional subscripts can be found from there by going back to the root and by collecting *upper subscripts* array entry. Figure 5 shows the logical arrangement of an SCEKA(n) along with necessary auxiliary tables.

4. Related Works

The chunking of multidimensional arrays is well addressed in the literature [19], [20] for MOLAP implementation. In this scheme the large multidimensional arrays are broken into chunks for better storage and processing. All the chunks are n dimensional with smaller length (of dimension) than the original array. To compress the array, a pair $\langle$ *ChunkNumber*, *OffsetInChunk*$\rangle$ is stored for each valid entry. The offset inside the chunk *OffsetInChunk* is computed using the multidimensional array linearization function. The idea is based on the traditional multidimensional array and dynamic extension is not possible. Compressed Row Storage (CRS) and Compressed Column Storage (CCS) [18], [20] are used due to their simplicity and purity with a weak dependence relationship between array elements in a sparse array. It uses 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 and CO stores the column (row for CCS) indices of those elements (for two dimensional arrays). For higher dimensional sparse arrays more one dimensional integer arrays are needed. Hence compression ratio and range of usability become impractical for higher dimensional arrays [5], [13], [21]. Ref. [17] proposes a multidimensional extendible array [22] based compression scheme, namely EaCRS which uses CRS scheme to compress each of the subarrays of the extendible array. For an n dimensional extendible array, the EaCRS scheme requires $n - 1$ auxiliary arrays for each of the $n - 1$ dimensional subarray to compress it. Hence the compression ratio is not good enough for higher number of dimensions. A compression scheme, namely ECRS/ECCS for array model EKMR [15] is presented in [16]. The scheme is based on CRS/CCS [18], [19] and applied on EKMR. The EKMR represents n dimensional arrays by a set of two dimensional arrays. Every two dimensional EKMRs map with a four dimensional traditional multidimensional array. Hence, when applying the CRS/CCS scheme on EKMR the number of auxiliary arrays is always less. Hence compression ratio and range of usability become efficient. But the CRS/CCS and ECRS/ECCS schemes are applicable for statically allocated arrays and are not suitable when the length of dimension and the number of dimension grow incrementally. A compression scheme for data warehouses using Hilbert curve [23] is proposed in [24] for statistically allocated arrays for predetermined size of dimension in the space and dynamic extension of the dimension size is not considered. [22], [25], [26] propose index array models for implementing extendible arrays, but do not provide any record encoding schemes or data compression technique. Since empty array elements occupy storage, they can be employed only for dense array; [25] handles the history tables using B tree structures and [26] uses axis vector to reduce the size of auxiliary tables. They are considered as sequence of the two consecutive extensions along the same dimension as an uninterrupted extension of that dimension and are handled by only one expansion record entry in the axial-vector. Therefore the number of elements in an axial vector is always less than or equal to the number of indices of the corresponding dimension. Hence it would be hard to apply the schemes to the actual storage organization especially for MOLAP.

5. Experimental Results

In this section we compare the performance of different array-based compression schemes, namely ECRS [15], [16], EaCRS [17], CRS [18]. The basic idea of these compressions is discussed in Section 4. The parameters that are assumed are described in Table 1. All the lengths and sizes are in bytes. The length of each dimension is equal, i.e. $l_i = l$, for all i. We placed the array in the secondary storage having the parameter values shown in Table 2 for all the schemes. We extend the length of dimensions of EKA in round robin fashion. All the tests are run on a machine (Dell Optiplex 380) of 2.932 GHz processor and 2 GB of main memory having a disk page size 4 KB using Microsoft C++. The auxiliary tables are stored in the main memory for faster access, since they act as an index to access the array elements.

5.1. Storage Cost

Figure 6 shows the amount of storage needed by different compression schemes for $n = 3, 4, 5$ and 6 respectively. The storage cost for CRS is always higher because it needs n auxiliary arrays to compress the n dimensional array. SCEKA always takes less amount of storage than CRS. SCEKA needs the same amount of storage as that of EaCRS only for $n = 3$; otherwise it needs less storage than EaCRS.

This is because EaCRS applies CRS technique for each $n-1$ subarray. Hence, when n increases, EaCRS shows bad performance. We see that ECRS is better than SCEKA in terms of storage required. In fact SCEKA demands a slightly higher amount of storage than ECRS, but be noted that SCEKA has the property of dynamic extendibility during run time, which is not possible in ECRS/ECCS without relocating the existing data. EaCRS is a dynamically extendible scheme, but requires more storage than SCEKA for $n \geq 4$.

l_i	Length of each dimension i, $l_i = l$, for all i
α	Size of each offset or auxiliary table cell
β	Size of each cell of the EKA
ρ	Density of the array. It is the ratio between non-empty array cells and total number of cells. $0 \leq \rho \leq 1$
η	Compression Ratio is defined as the ratio between the compressed array and that of uncompressed array. The value of η is preferred to be $0 < \eta < 1$.
υ	Range of Usability of compression scheme and is defined as the maximum value of ρ up to which the compression ratio (η) is less than 1.
n	No. of dimensions of the array.
λ	The length of extension in each dimension.
NRQ	Number of subscripts selected for range key query.
$\max(l_i)$	Maximum length of a specific dimension i for fixed n.
$\min(l_i)$	Minimum length of a dimension i for specific values of n.

Table 1. Parameters for SCEKA.

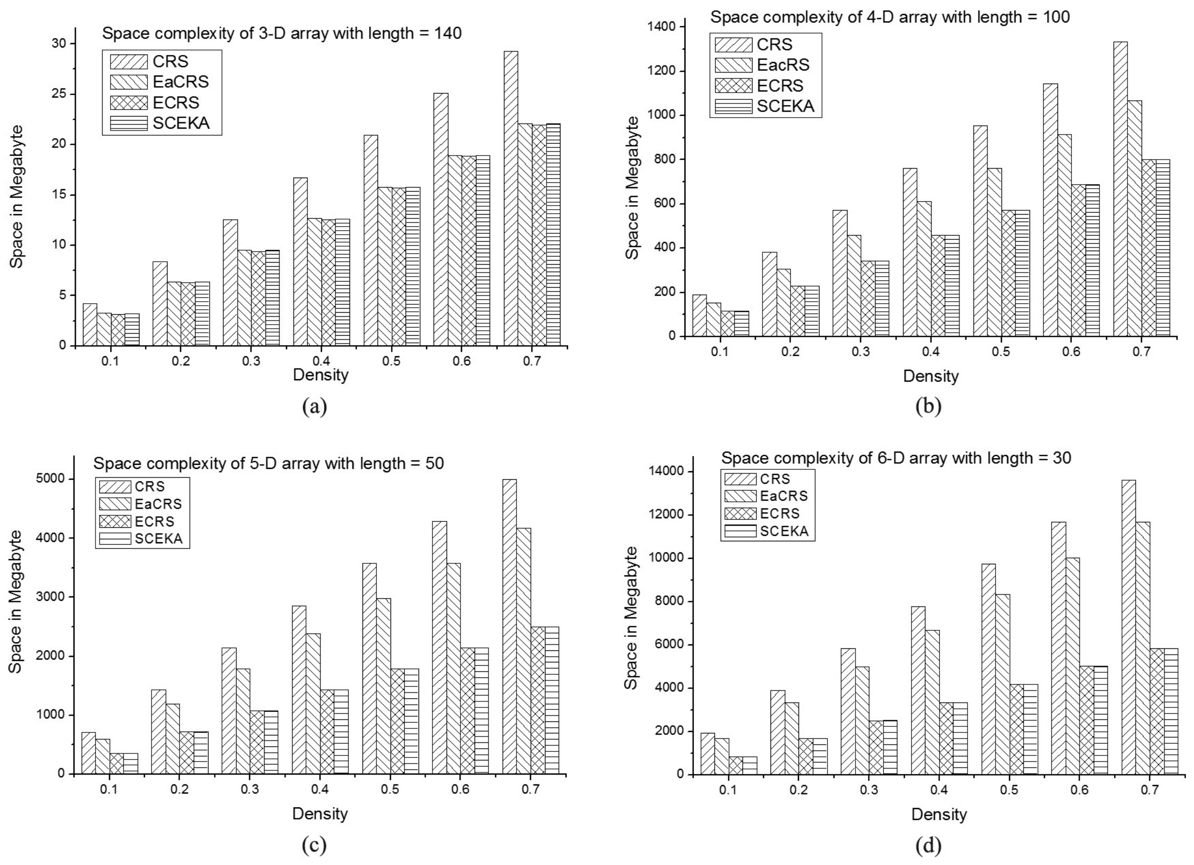


Figure 6. Comparison of storage requirement for different compression schemes.

n	α	β	λ	$\max(l_i)$	$\min(l_i)$	NRQ
3-6	4	8	2-20	30-140	10-40	$(l-\lambda)/2$ to $(l+\lambda)/2$

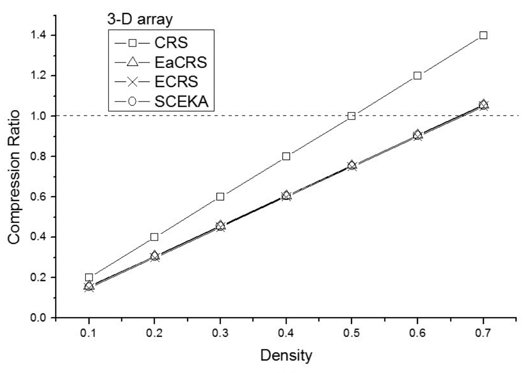
Table 2. Parameters for constructed prototypes.

5.2. Compression Ratio and Range of Usability

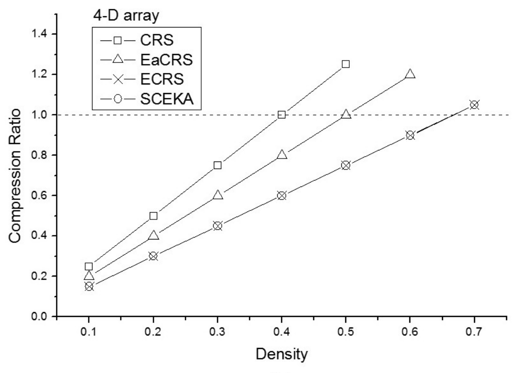
Figure 7 shows the comparison of the ranges of usability in different compression schemes for varying n. SCEKA and ECRS cross the range of usability line at an approximate value of $\rho = 0.66$, which is better than or equal to CRS or EaCRS scheme. Figure 7(e) represents the compression ratio with constant density of 0.4. We find that CRS and EaCRS are usable up to $n = 3$ and 4 respectively, for $\rho = 0.4$, but SCEKA can be of any number of dimensions. This is because the compression ratio of SCEKA does not depend on the number of dimensions.

5.3. Extension Cost

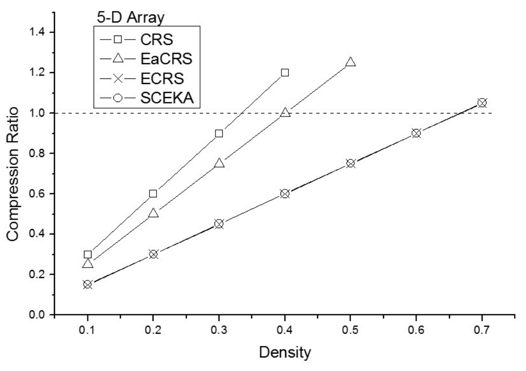
Figure 8 shows the extension time of SCEKA(4) and EKA(4) for different values of ρ. For SCEKA(4) (Figure 8(a)) the extension time varies with density. On the other hand EKA(4) always takes similar time for extension, irrespective of the values of ρ (shown in Figure 8(b)). This is because in uncompressed version of EKA, the density of real data does not affect the total size of the extension subarray as disk I/O is constant. Figure 8(c) and 8(d) show the average extension time taken by SCKEA and EKA for $n = 4$ and 5 respectively. In both cases, SCEKA requires less time than EKA, the reason is subtle, the compressed array needs less data to write, hence fewer disks I/O are required and therefore time is less. We see that the difference in extension time increases with the increase in length of dimension.



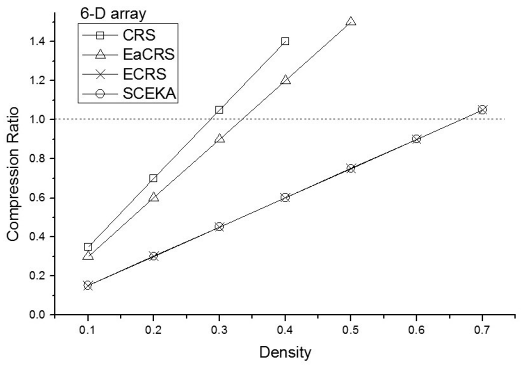
(a)



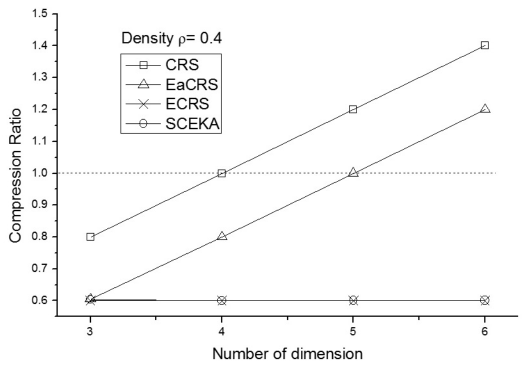
(b)



(c)



(d)



(e)

Figure 7. Comparison of the range of usability of different compression schemes.

5.4. Retrieval Cost

Figure 9 presents the average retrieval performance for range key queries of NRQ subscripts on both SCEKA(4) and EKA(4) with different values of ρ. The retrieval time varies with ρ in SCEKA(4). However, there is no effect of ρ in uncompressed EKA. The retrieval time is almost constant for a particular length of dimension (Figure 9(b)). This is because in EKA(4), the segment or subarray sizes (that are read from storage) remain the same irrespective of the values of ρ for uncompressed EKA, hence the retrieval performance is constant. Similar results are also found for EKA(5) and EKA(6). Figures 9(c) and 9(d) show the comparison of average range key retrieval time of NRQ subscripts in SCEKA and EKA for $n = 5$ and 6, respectively for $\rho = 0.4$, 0.5, and 0.6, respectively. In every case the SCEKA needs less time than EKA. The reason is that in uncompressed EKA, whatever the density, the segment is always the same and the retrieval time is higher. Furthermore, if the density is less than 1, we need a linear search to be made for determining the non-empty cells. On the other hand, in compressed

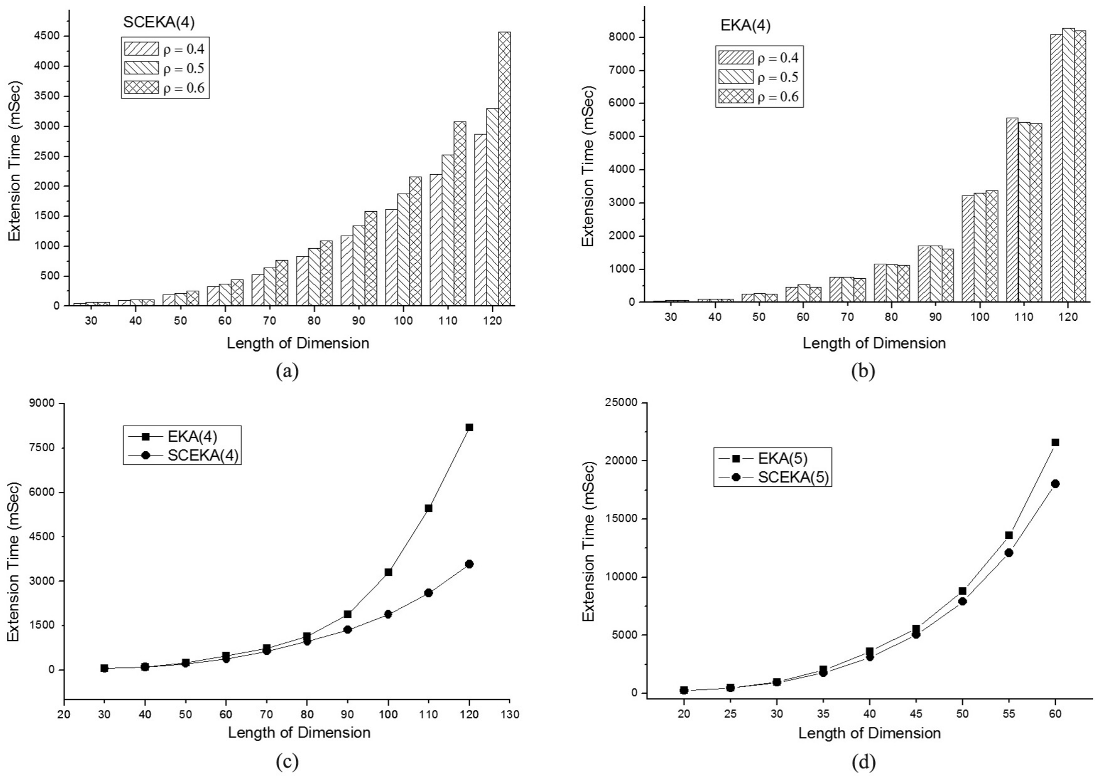


Figure 8. Extension time comparison of SCEKA and EKA.

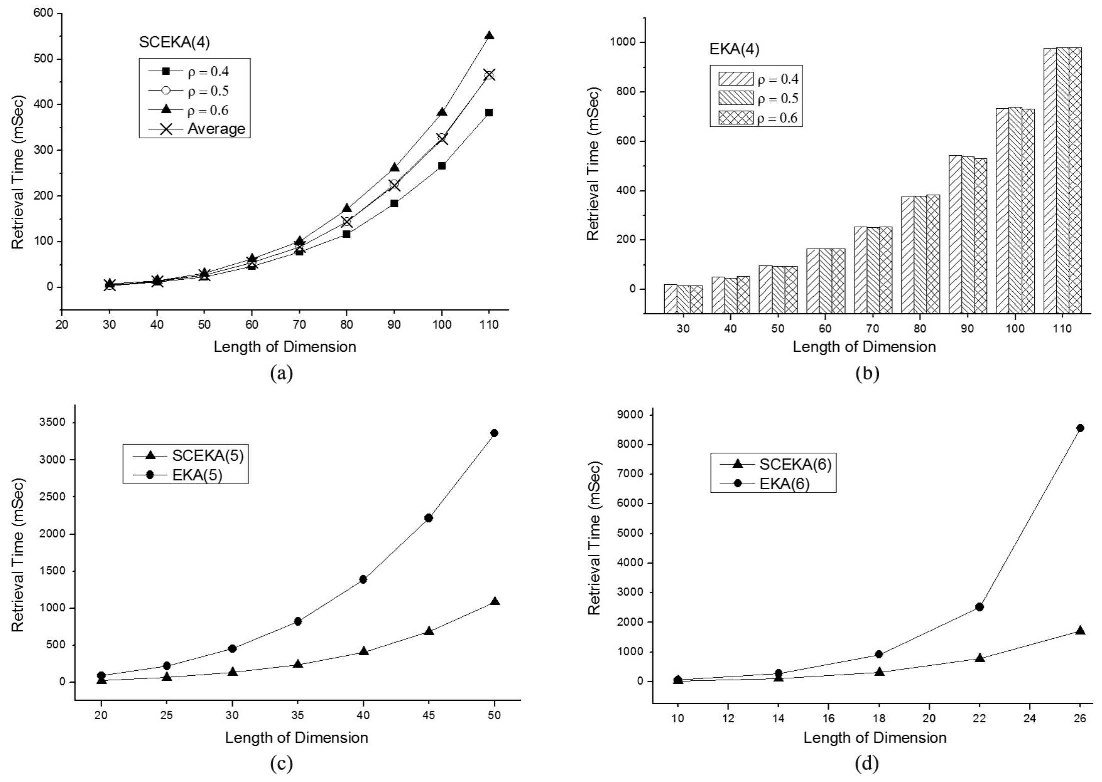


Figure 9. Average range key retrieval time of SCEKA and EKA.

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 for any search. Simply read the segment from the disk to show the result. This takes less time. Therefore, overall retrieval time in SCEKA is better than that in uncompressed EKA.

6. Conclusion

Multidimensional arrays are extensively used in many scientific applications to represent their data for efficient processing. However, in many situations the total number of dimension and length of dimension cannot be predicted. Besides this, representing the real world data in multidimensional array creates a very sparse array. In this paper, we proposed a new compression scheme that can grow dynamically during runtime. We compared the scheme with various other schemes for their practical usability. The proposed model is very efficient when dynamic extension of the array is an issue. We believe that the proposed scheme can be successfully applied to database applications, especially for multidimensional databases or multidimensional data warehousing systems. One important future direction of the work is that the scheme can easily be implemented in parallel platforms. Because most of the operations described here are independent of each other, it will be very efficient to apply this scheme in parallel and multiprocessor environments.

References

[1] S. Sarawagi and M. Stonebraker, Efficient organization of large multidimensional arrays. In *Proceedings of 10th International Conference on Data Engineering (ICDE'94)*, pp. 328–336, 1994.

[2] M. S. Mit, J. B. Slac, D. D. Microsoft, K. tat Lim, S. Zdonik, Requirements for Science Data Bases and SciDB. In: Proc. of Conference on Innovative Data, Systems Research, CIDR'09.

[3] S. M. M. Ahsan and K. M. A. Hasan, An Efficient Encoding Scheme to Handle the Address Space Overflow for Large Multidimensional Arrays. *Journal of Computers*, **8**(5), pp. 1136–1144, May 2013.

[4] E. J. Otoo, G. Nimako, D. OheneKwofie, Chunked extendible dense arrays for scientific data storage. *Parallel Computing*, **39**(12), 802–818, 2013.

[5] K. M. A. Hasan, M. Kuroda, N. Azuma, T. Tsuji, and K. Higuchi, An extendible array based implementation of relational tables for multi dimensional databases. In*Proceedings of 7th International Conference on Data Warehousing and Knowledge Discovery (DaWaK'05)*. Denmark: LNCS 3580, Springer Berlin Heidelberg, pp. 233–242, 2005.

[6] S. Joannou, R. Raman, An Empirical Evaluation of Extendible Arrays. In *Proceedings of 10th International Symposium on Experimental Algorithms (SEA'11)*, Kolimpari, Greece, pp. 447–458, 2011.

[7] K. M. A. Hasan, T. Tsuji, and K. Higuchi, An Efficient Implementation for MOLAP Basic Data Structure and Its Evaluation. In*Proceedings of 12th International Conference on Database Systems for Advanced Applications (DASFAA'07)*, pp. 288–299, 2007.

[8] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh, Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals, *Microsoft Research*, TechReport MSR-TR-95-22, 1995.

[9] S. M. M. Ahsan and K. M. A. Hasan, An Implementation Scheme for Multidimensional Extendable Array Operations and Its Evaluation. In*Proceedings of International Conference on Informatics Engineering & Information Science (ICIEIS'11)*. Kuala Lumpur, Malaysia: CCIS 253, Springer Berlin Heidelberg, pp. 136–150, 2011.

[10] S. M. M. Ahsan and K. M. A. Hasan, Extendible Multidimensional Array Based Storage Scheme for Efficient Management of High Dimensional Data, IJNGC **4**(1), pp. 88–105, 2013.

[11] M. M. Mano, *Digital Logic and Computer Design*. Prentice Hall, 2005.

[12] L. Jianzhong, J. Srivastava, Efficient aggregation algorithms for compressed data warehouses. *IEEE Transactions on Knowledge and Data Engineering*, **14**(3), pp. 515–529, May 2002.

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

[14] K. Kaczmarski, T. Rudny, MOLAP cube based on parallel scan algorithm. In *Proceedings of 15th International Conference on Advances in Databases and Information Systems (ADBIS'11)*. Vienna, Austria: LNCS 6909, Springer Berlin Heidelberg, pp. 125–138, 2011.

[15] Chun-yuan Lin, Jen-shiuh Liu, and Yeh-ching Chung, Efficient representation scheme for multidimensional array operations. *IEEE Transactions on Computers*, **51**(3), pp. 327–345, Mar. 2002.

[16] CHUN-YUAN LIN, YEH-CHING CHUNG, AND JEN-SHIUH LIU, Efficient data compression methods for multidimensional sparse array operations based on the EKMR scheme. *IEEE Transactions on Computers*, **52**(12), pp. 1640–1646, Dec. 2003.

[17] R. ISLAM, K. M. A. HASAN, AND T. TSUJI, EaCRS: an extendible array based compression scheme for high dimensional data. In *Proceedings of the Second Symposium on Information and Communication Technology (SoICT'11)*, Hanoi, Vietnam, pp. 92–99, 2011.

[18] R. BARRETT, ET AL., *Templates for the solution of linear systems: Building blocks for iterative methods*, 2nd ed. Philadelphia, PA: SIAM, 1994.

[19] D. ROTEM, E. J. OTOO, S. SESHADRI, Chunking of Large Multidimensional Arrays, Lawrence Berkeley National Laboratory, University of California LBNL-63230, 2007.

[20] Y. ZHAO, P. M. DESHPANDE, J. F. NAUGHTON, An array-based algorithm for simultaneous multidimensional aggregates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '97)*, pp. 159–170, 1997.

[21] K. M. A. HASAN, Compression schemes of high dimensional data for MOLAP. In *Evolving Application Domains of Data Warehousing and Mining: Trends and Solutions*, P. FURTADO, ED. Information Science Reference, Hershey, PA, ch. IV, pp. 64–81, 2010.

[22] E. J. OTOO AND T. H. MERRETT, A storage scheme for extendible arrays. *Journal of Computing*, **31**,(1), pp. 1–9, 1983.

[23] J. K. LAWDER, P. J. H. KING, Querying multi-dimensional data indexed using the Hilbert space-filling curve. *ACM SIGMOD Record*, **30**(1), pp. 19–24, Mar. 2001.

[24] T. EAVIS, D. CUEVA, A Hilbert Space Compression Architecture for Data Warehouse Environments. In *Proceedings of 9th International Conference on Data Warehousing and Knowledge Discovery (DaWaK'07)*, Regensburg Germany, pp. 1–12, 2007.

[25] D. ROTEM, J. L. ZHAO, Extendible arrays for statistical databases and OLAP applications. In *Proceedings of 8th International Conference on Scientific and Statistical Database Systems*, pp. 108–117, 1996.

[26] E. J. OTOO, D. ROTEM, Efficient Storage Allocation of Large-Scale Extendible Multi-dimensional Scientific Datasets. In *Procedings of 18th International Conference on Scientific and Statistical Database Management (SSDBM'06)*, pp. 179–183, 2006.

Received: July, 2014
Accepted: November, 2014

Contact addresses:

Sk. Md. Masudul Ahsan
Dept. of Computer Science and Engineering
Khulna University of Engg.& Tech.(KUET)
Bangladesh
e-mail: `masudul.ahsan@gmail.com`

K. M. Azharul Hasan
Dept. of Computer Science and Engineering
Khulna University of Engg.& Tech.(KUET)
Bangladesh
e-mail: `azhasan@gmail.com`

SK. MD. MASUDUL AHSAN 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 a PhD student in Kyushu Institute of Technology, Japan. His current research interests lie in the fields of multi-dimensional database implementation schemes, visual modeling, and machine vision. He has been a faculty member of the Department of Computer Science and Engineering, Khulna University of Engineering and Technology (KUET), Bangladesh since 2004.

K. M. AZHARUL HASAN received his B.Sc. (Engg.) degree 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 multidimensional databases, information retrieval and high performance computing. He has been with the Department of Computer Science and Engineering at Khulna University of Engineering and Technology (KUET), Bangladesh since 2001.